

**RUNTIME MONITORING OF C PROGRAMS
FOR SECURITY AND CORRECTNESS**

by

Suan Hsi Yong

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2004

© Copyright by Suan Hsi Yong 2004

All Rights Reserved

ABSTRACT

Finding errors in software is a difficult problem: millions of dollars are spent in testing and debugging, yet latent bugs continue to be discovered even in thoroughly tested programs. Apart from the undesirable effects of producing incorrect results, crashing systems, and corrupting data, bugs also raise security concerns: memory-safety errors like buffer overruns and stale pointer dereferences can be exploited by malicious agents to acquire confidential data or seriously compromise the target system, sometimes in undetectable ways.

A key feature of the C programming language is that it allows low-level control over memory usage and runtime behavior; this flexibility is crucial for many programming settings, such as at the system level, and is one reason why C programs continue to be widely used today. However, this flexibility is achieved at the price of safety: the language syntax is too weak to prevent type errors and memory-safety errors from occurring at runtime.

This work explores three related approaches to detect errors in C programs via runtime monitoring. The Memory-Safety Enforcer is a tool that detects memory-safety errors at runtime, and can be used for both security and debugging. The Sensitive Location Checker is a security tool that prevents invalid memory accesses from overwriting sensitive locations that are vulnerable to attack. The Runtime Type Checker is a debugging tool for detecting bugs that manifest themselves as type errors at runtime. All three approaches tag memory locations with auxiliary information at runtime, and make use of static analysis to improve performance and coverage by eliminating unnecessary runtime instrumentation.

ACKNOWLEDGMENTS

I wish to thank the countless people — friends, colleagues, advisors, instructors — who have made my years at Madison the most enriching and enjoyable of my life. Special thanks go to my advisor, Susan Horwitz, whose guidance and tireless support have proven invaluable in my journey through Graduate School. Others with whom I have had the pleasure of collaborating include Manuvir Das, Alexey Loginov, and Thomas Reps, and the small circle of PL students at the University of Wisconsin. Thanks go out also to my committee, comprising Susan and Tom as well as Charles Fischer, Somesh Jha, and James Smith, for their invaluable feedback.

I also wish to thank my family back in Penang, Malaysia, whose support and encouragement are never in doubt.

This work was supported in part by the National Science Foundation under grants CCR-9970707, CCR-9987435, and CCR-0305387.

TABLE OF CONTENTS

| | Page |
|---|------|
| ABSTRACT | i |
| 1 Introduction | 1 |
| 1.1 Overview | 1 |
| 1.2 Memory-Safety Enforcer | 4 |
| 1.3 Sensitive Location Checker | 7 |
| 1.4 Runtime Type Checker | 8 |
| 1.5 Design Overview | 10 |
| 1.6 Thesis Outline | 12 |
| 2 Control-Transfer Attacks | 13 |
| 2.1 Buffer-Overflow Vulnerabilities | 15 |
| 2.2 Format-String Vulnerabilities | 16 |
| 2.3 Erroneous Free | 18 |
| 2.3.1 Traceroute Vulnerability | 18 |
| 2.3.2 GNU Memory Management | 19 |
| 2.3.3 The Exploit | 21 |
| 3 Memory-Safety Enforcer (MSE) | 23 |
| 3.1 Memory-Safety Model | 24 |
| 3.2 Fat Pointers Approach | 26 |
| 3.3 Tagged Memory Approach | 28 |
| 3.4 MSE Classification Framework | 30 |
| 3.4.1 Naive Classification | 32 |
| 3.4.2 Read-Write vs. Write-Only Checking | 33 |
| 3.5 Improving MSE | 34 |
| 4 Classifications Using Points-To Analysis | 36 |
| 4.1 Classifying Tracked Locations | 37 |

| | Page | |
|----------|--|-----------|
| 4.2 | Extended Points-To (EPT) Analysis | 38 |
| 4.2.1 | May-Be-Uninitialized Pointers | 41 |
| 4.2.2 | Performance | 42 |
| 4.3 | Stack and Heap Locations | 44 |
| 5 | Redundant Checks Analysis | 46 |
| 5.1 | Affecting Locations | 47 |
| 5.2 | Evaluation | 52 |
| 6 | Pointer-Range Analysis | 55 |
| 6.1 | Representing Ranges | 56 |
| 6.2 | Dataflow Facts and Functions | 61 |
| 6.3 | Evaluating Expressions | 62 |
| 6.3.1 | Well-Typed Arithmetic | 64 |
| 6.3.2 | Mismatched-Type Arithmetic | 65 |
| 6.4 | Predicates | 69 |
| 6.5 | Widening and Narrowing | 71 |
| 6.6 | Structure Fields | 73 |
| 6.7 | Evaluation | 75 |
| 7 | Summary of Memory-Safety Enforcer | 80 |
| 7.1 | Performance and Coverage | 80 |
| 7.2 | Comparison with Other Tools | 84 |
| 7.3 | Effectiveness of the MSE | 85 |
| 7.3.1 | Fault-Injection Study | 85 |
| 7.3.2 | Finding Bugs | 87 |
| 7.3.3 | Detecting Attacks | 87 |
| 7.4 | Conclusion | 89 |
| 8 | Sensitive Location Checker (SLC) | 90 |
| 8.1 | Effectiveness of SLC vs. MSE | 92 |
| 8.2 | Library Functions | 93 |
| 8.3 | Implementation | 94 |
| 8.4 | Evaluation | 96 |
| 8.5 | Summary: SLC vs. MSE | 99 |

| | Page |
|---|------|
| 9 Runtime Type Checker (RTC) | 101 |
| 9.1 Motivating Examples | 102 |
| 9.1.1 Bad Union Access | 102 |
| 9.1.2 Custom Allocator | 103 |
| 9.1.3 Simulating Inheritance with Structures | 104 |
| 9.2 Type Safety | 106 |
| 9.2.1 C Language and Static Type System | 107 |
| 9.2.2 Runtime Type-Safety Model | 108 |
| 9.3 Tracking Runtime Types | 111 |
| 9.4 Instrumentation | 113 |
| 9.4.1 Library Functions | 117 |
| 9.5 Experience with Finding Bugs | 117 |
| 9.5.1 False Positives | 119 |
| 9.6 Performance Evaluation | 120 |
| 10 Improving the Runtime Type Checker | 122 |
| 10.1 Type-Flow Analysis | 122 |
| 10.1.1 Analysis Outline | 126 |
| 10.1.2 Building the Assignment Graph | 126 |
| 10.1.3 Computing Possible and Expected Types | 128 |
| 10.1.4 Categorizing Expressions and Locations | 130 |
| 10.1.5 Experimental Results | 134 |
| 10.2 Stack and Heap Locations | 134 |
| 10.3 May-Be-Uninitialized Analysis | 137 |
| 10.4 Redundant Checks Analysis | 141 |
| 10.5 Summary of RTC Optimizations | 143 |
| 11 Related Work | 146 |
| 11.1 Security-Oriented Approaches | 146 |
| 11.2 Memory and Type Safety | 148 |
| 11.2.1 Fat Pointers | 148 |
| 11.2.2 Tagged Memory | 151 |
| 11.2.3 Runtime Type Checking | 152 |
| 11.2.4 Other Runtime Monitoring Ideas | 152 |
| 11.3 Eliminating Array-Bounds Checks | 153 |
| 11.4 Static Error Detection | 154 |

| | Page |
|--|------|
| 12 Conclusion and Future Directions | 155 |
| APPENDIX Summary of Benchmarks | 158 |
| LIST OF REFERENCES | 163 |
| INDEX | 171 |

Chapter 1

Introduction

1.1 Overview

Writing error-free software remains an elusive goal; in fact, it has become an accepted truism in the software industry that software can never be 100% bug-free. Despite millions of dollars being spent on software testing, latent errors remain in programs and can go undetected for long periods of time.

There are two main obstacles that make finding software errors a difficult task. Firstly, each program defines a very large, usually infinite, number of possible states, making it impossible to test exhaustively. In fact, as software grows larger and more complicated, the proportion of the state space that can be effectively tested decreases. Secondly, even knowing what symptoms to look for is a non-trivial task, since an error may not change the observable output of the program, or may change it in a way that is difficult to recognize, either automatically or by a human.

So how important is it to fix errors in software, especially if the error resides in a seldom-executed corner of the state space, or does not noticeably affect the output? There are some critical applications, such as medical life-support, business accounting, or space exploration, for which any software failure can lead to drastic or expensive consequences. Additionally, as more and more new software is built on top of old legacy systems, it becomes increasingly important for the underlying legacy system to be correct and error-free. Furthermore, even in application software where the only consequence of a software failure may be user

inconvenience, a seldom-exercised error can become a serious security vulnerability. Internet worms like the Morris worm, Code Red, and Slammer exploit software bugs that would not be exercised under normal usage, and have cost the software industry billions of dollars [Moo⁺02].

One way to improve software quality is to impose restrictions at the programming-language level. Concepts like *memory safety* and *type safety* have proven to be effective at making programs easier to understand and maintain. However, there is a tradeoff in enforcing memory and type safety: if a language is too restrictive, it may limit the programmer's ability to program effectively. Further, enforcing memory and type safety may incur a high execution-time overhead.

C and C++ are two languages that mandate memory safety and type safety, but do not strictly enforce them. This means that the programmer, rather than the compiler, is responsible for ensuring that memory and type safety are observed, and that the programmer may circumvent memory and type safety, either deliberately or inadvertently. While this weak enforcement of safety is what enables these languages to be both flexible and efficient — qualities that contribute to their continued popularity and widespread use — it also makes certain classes of errors difficult to detect, and accounts for a large proportion of security vulnerabilities in today's software [Cow⁺98, Wag⁺00].

A number of approaches have been proposed for enforcing memory safety in C programs. However, none of these approaches have been able to maintain both the flexibility and efficiency of the language. Static analysis has limited power: only a very restricted subset of C programs can be guaranteed never to violate memory safety [Dhu⁺03], unless additional programmer annotations are supplied [DeL⁺01]. Dynamic approaches, on the other hand, either incur a high runtime overhead [Aus⁺94, Jon⁺97] or must add restrictions to the language — in particular, to its heap memory management semantics [Jim⁺02, Con⁺03].

Enforcing type safety is an even more challenging task. The prevalent use of casting in C can make programs difficult to understand and susceptible to subtle errors [Sif⁺99]. While

there have been many proposals for alternative type systems to apply to C, there have been few proposals for enforcing type safety for C programs at runtime.

This thesis describes an approach that uses *tagged memory* to enforce memory safety and type safety at runtime. The underlying idea is to tag each byte of memory with extra bits of information at runtime, and to instrument certain operations to check for safety violations. The tags are stored in a *mirror memory*, which introduces a constant slowdown factor for looking up tags, making the approach scalable to programs with large memory footprints. Instrumentation of runtime checks is performed at the source level, which enhances portability and allows the approach to take advantage of type information available at the source level. The approach has been implemented for the C language, but the ideas are applicable to any language that does not strictly enforce memory or type safety, like C++ and other derivatives of C.

An important aspect of this work is the use of static analysis to improve both precision and efficiency. The idea is to determine at compile time which statements are guaranteed to be safe, and to eliminate runtime checks for those statements. Various analyses are explored, from highly-scalable flow-insensitive approaches to more expensive flow-sensitive ones, to evaluate the tradeoffs between improved runtime performance and slower analysis times.

Three manifestations of the tagged memory idea are evaluated in this thesis. The Memory-Safety Enforcer (MSE) checks for memory-safety violations in deployed software to prevent malicious attacks from exploiting vulnerabilities; the Sensitive Location Checker (SLC) is a variant of the MSE that prevents only specific sensitive locations from being corrupted by invalid writes; the Runtime Type Checker (RTC) checks for correct usage of types at runtime, and may be used during program development to detect subtle errors that would not be detected by other approaches.

1.2 Memory-Safety Enforcer

Memory safety can be loosely defined as the condition in which each dereference (pointer indirection or array-index expression) accesses memory that is within the bounds and lifetime of its intended referent. A memory-safety violation occurs when a dereference attempts to access memory beyond the bounds of its intended referent, or attempts to access an object after it has been deallocated. The Memory-Safety Enforcer (MSE) instruments a C program with code to check for memory-safety violations at runtime: if a violation is about to occur, it terminates the program to prevent any potential malicious attacks from exploiting the violation. The MSE is intended for use in deployed software, which means it must be sound (it should report no false positives) and efficient (it should have a low runtime overhead). These goals are achieved while being compliant with the C language and preserving its flexibility, thus allowing the approach to be applied automatically to legacy programs.

The basic idea behind MSE is to tag each byte of memory with one bit to indicate whether it is a *valid* target of a dereference. The instrumented program sets the tag of each object to *valid* when it is allocated, and *invalid* when it is deallocated; it also checks each dereference to see whether the tags of its target are *valid*. If a dereference is about to access memory that is tagged *invalid*, a memory-safety violation is reported and the program is halted.

Figure 1.1 illustrates this approach; it shows a piece of code that is vulnerable to a stack-smashing attack. The code copies a given string (pointed to by `src`) into a buffer (`buf`) that is allocated on the stack, without checking for overflow. By providing a string longer than 16 bytes, a user of this code can exploit this flaw to overwrite the return address on the activation record with the address of the user's own code (stored, for example, in a command-line argument). In this example, only the bytes corresponding to variable `buf` can be validly accessed by a dereference; thus, as shown in Figure 1.1(c), only those bytes are tagged *valid*. This means that the first attempt to write into a location outside the `buf` array will be detected, and the attack will be prevented.

MSE can instrument a program in one of two modes: write-only and read-write. The former checks only for *writes* that violate memory safety while the latter checks for both reads and writes. Write-only checking is an attractive option for security because it incurs a significantly lower runtime overhead without sacrificing much protection, since, as far as we know, the most dangerous class of attacks always requires a *write* safety violation to succeed. Read-write checking may be desirable in certain settings, since *read* violations may still be exploited to gain access to confidential data.

Naively tagging all user-defined locations as *valid* and checking all dereferences has two drawbacks: it has a high runtime overhead and it may miss detecting some violations. For example, consider the code shown in Figure 1.2. This code is similar to that of Figure 1.1, but in addition to copying the given string into the `buf` array, it keeps track of the number of characters copied, writing that value into location `x` via a dereference of pointer `p`. In this example, since both `buf` and `x` may be validly accessed via dereferences, if the copied string overflows `buf` and overwrites `x`, that invalid access will not be detected; only if the location below `x` on the stack (which will be tagged *invalid*) is overwritten will the buffer overflow be detected.

Note however that in fact it is not necessary to check the dereference of `p` at runtime; that dereference can *never* cause an invalid access, since `p` can only point to `x`. Furthermore, since `p` is the only pointer that can legitimately point to `x`, if dereferences of `p` are not checked, there is no need to tag `x` *valid*, and in that case the invalid access via `*dst` that overwrites `x` can be detected.

This motivates an important insight: if a dereference can be statically guaranteed to be safe (i.e., unable to cause an invalid access), then the dereference need not be checked at runtime. Furthermore, if a location can only be accessed (legitimately) either directly or via unchecked dereferences, then the location need not be tagged *valid*. This can lead both to lower runtime overhead and to the detection of more invalid accesses as illustrated by the example in Figure 1.2.

To apply this idea, static analysis is used to classify each dereference as either *definitely safe* or *potentially unsafe*, and to classify each location as either *tracked* or *untracked*. A dereference is definitely safe if it can never cause a memory-safety violation; otherwise it is potentially unsafe. Only a dereference that is potentially unsafe needs to be checked for validity at runtime; definitely safe dereferences need not be checked. A location is classified as tracked if it may be legitimately accessed by a potentially unsafe dereference. Since definitely safe dereferences are not checked for validity, untracked locations need not be tagged as *valid* at runtime. This means that identifying fewer tracked locations will increase the likelihood of detecting a memory-safety violation. Therefore, the goal of static analysis is to identify as few potentially unsafe dereferences and tracked locations as possible while maintaining soundness (no false positives) and coverage (no new false negatives). Improvements in static analysis will lead to a dual gain: better runtime performance (due to fewer runtime checks) and better coverage (increased likelihood of detecting a violation).

A number of static analyses of varying precision and complexity are used to improve the performance and coverage of MSE. *Points-to Analysis* is used to compute the set of tracked locations as the legitimate targets of potentially unsafe dereferences, and is extended to identify an approximate set of definitely safe dereferences. *Redundant Checks Analysis* identifies dereferences for which a runtime check would be redundant. *Pointer Range Analysis* computes the range of possible values for each variable, to identify dereferences that are guaranteed to be in-bounds.

1.3 Sensitive Location Checker

A variation of the MSE, called the Sensitive Location Checker (SLC), limits the runtime checks to preventing corruption of specific “sensitive” locations in memory. These sensitive locations include the return address field in each activation record, function pointers, arguments to the `system` and `exec` functions, and other locations that are vulnerable to known methods of attack.

The approach tags sensitive locations as *invalid* at runtime, and checks potentially unsafe dereferences to make sure they do not overwrite a sensitive location that is tagged *invalid*. Static analysis is used to identify dereferences that may legitimately write to sensitive locations; these dereferences are not checked (as they would otherwise report a false positive). With fewer locations needing to be explicitly tagged at runtime, this approach can yield significant runtime performance improvements over the MSE, and can detect some attacks that would be missed by the MSE (though, on the other hand, there are also attacks that would be detected by the MSE but not by the SLC.)

1.4 Runtime Type Checker

The Runtime Type Checker (RTC) instruments a C program to check for consistent use of types at runtime. It detects errors that manifest themselves as type errors, including subtle bugs that would not be detected by other approaches. It also reports warnings when it encounters suspicious type behavior that may not be an actual error; these warnings are often useful for diagnosing the root cause of subsequently reported errors.

The underlying type-safety model is an important consideration in the design of a runtime type checker for C. Type safety is sometimes circumvented intentionally by programmers for efficiency or generality; such programming practices may trigger too many false positives if the type-safety model is too strict. The model adopted by RTC derives type information from *values* rather than *locations* (i.e., declared types of memory locations are ignored), and checks the type of a value only when it is *used* in a given typed context. An assignment is treated not as a use, but as an untyped memory-copy operation.

At runtime, the RTC tags each byte of memory with a four-bit tag to encode the type of the data stored in that byte. Only scalar types are tracked, and all pointer types are treated as compatible with each other. For an aggregate object, its component scalars are tracked individually. When the value in a memory location is *used* in the context of a given type, if the tag of that location does not match the type of the use, a type error is reported.

Further, for each assignment, if the assigned value does not match the declared type of the assignment, a warning message is issued to indicate suspicious type behavior.

| | i's tag | p's tag |
|--------------------------------|---------------|----------------|
| 1. <code>int i;</code> | <i>uninit</i> | <i>unalloc</i> |
| 2. <code>int *p;</code> | <i>uninit</i> | <i>uninit</i> |
| 3. <code>i = 10;</code> | <i>int</i> | <i>uninit</i> |
| 4. <code>p = (int *) i;</code> | <i>int</i> | <i>int</i> |
| 5. <code>*p = 0;</code> | <i>int</i> | <i>int</i> |

Figure 1.3

Figure 1.3 presents an illustrative example. The two columns on the right give the RTC tags associated with locations `i` and `p` after each statement is executed: each memory location is tagged *unallocated* initially, and tagged *uninitialized* when it is allocated. At line 3, an integer value is written into location `i`, causing `i` to be tagged *int*. At line 4, the value in `i` is copied into the pointer `p` via an explicit type cast; this causes the tag of `i` (*int*) to be copied into the tag of `p`. Since this tag (*int*) does not match the static type of the assignment (`int *`), a warning message is issued to indicate suspicious type behavior. Finally, at line 5, the dereference of `p` is a pointer-typed use of the value in `p`; this causes the RTC to check that the tag of `p` is *pointer*: if it is not (as in the example), a type error is reported.

As may be expected, the runtime overhead of RTC is quite high. While this may be acceptable in a debugging or testing setting, it can be significantly improved with static analysis. The first improvement, called *Type-Flow Analysis*, computes (conservatively) the set of runtime types that may be associated with each *lvalue* expression: if the computed runtime type for a given expression is the same as its expected type, then the expression is guaranteed to be type-safe, and does not need to be checked at runtime. The second improvement is *Redundant Checks Analysis* which identifies and eliminates redundant RTC checks.

1.5 Design Overview

The MSE, SLC, and RTC share a common architecture, which is depicted in Figure 1.4. The core functionality follows the track on the left and bottom edge of the figure: each C source file is fed to the instrumenter to generate an instrumented source file that includes calls to tag-management routines. The instrumented source file is compiled with a regular C compiler, and linked with the MSE, SLC, or RTC runtime library to generate the instrumented executable. The static-analysis phase, depicted in the upper-right corner, analyzes the original C source files and supplies a classification to determine which expressions in the program need to be instrumented with runtime checks, and which expressions may have their runtime checks elided.

Our approach has the following advantages over other approaches:

1. **Portability:** By instrumenting programs at the source level, the implementation can be applied on any platform with an ANSI C compiler. This is in contrast to approaches that instrument object code or that are tightly coupled with a specific compiler.
2. **Modularity:** The core approach allows separate compilation; that is, different source files may be instrumented and compiled independently before being linked. The only restriction is in the static analysis, which, to be sound, may require access to all the source files.
3. **Compatibility:** The mirror implementation leaves the program's data representations unchanged, which allows for compatibility with libraries and uninstrumented modules. This makes it easy to apply the approach to legacy code, or to instrument only selected components of a larger system. However, to preserve soundness, the behavior of uninstrumented components must be properly accounted for by a wrapper routine as well as by the static analysis.
4. **Flexibility:** The tagged-memory approach preserves the flexibility of the C language, allowing explicit memory management (i.e., without requiring the use of a garbage

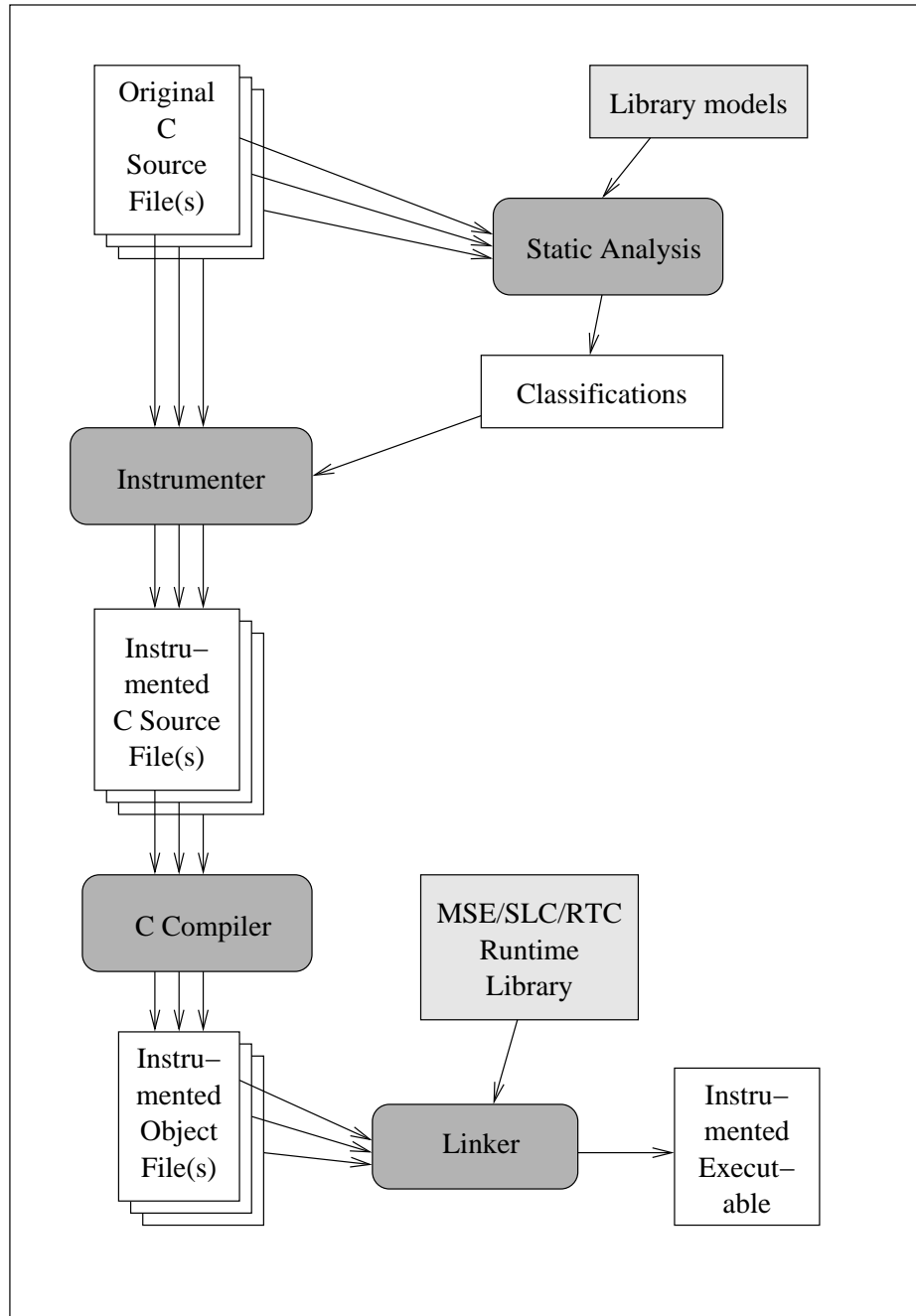


Figure 1.4 MSE/RTC Architecture.

collector) as well as specialized programming methods like custom allocators, and unorthodox practices like casting between pointers and integers.

5. Scalability and Efficiency: The mirror implementation of tagged memory introduces a linear space overhead and a constant execution time overhead for each runtime check, making the approach scale to large programs.
6. Protected monitor data: The mirror implementation of the tags are protected from being corrupted (e.g., by a malicious agent), because the mirror memory locations themselves are tagged *invalid* or *unallocated*.

1.6 Thesis Outline

This thesis begins by describing, in Chapter 2, a class of security attacks called Control Transfer Attacks, to motivate the use of the MSE and SLC. Next, an overview of the MSE is given in Chapter 3, followed by a description of various static-analysis classification schemes to improve the performance and coverage of the MSE: Points-to Analysis (Chapter 4), Redundant Checks Analysis (Chapter 5), and Pointer-Range Analysis (Chapter 6). Chapter 7 evaluates and summarizes the MSE.

Chapter 8 describes and evaluates the SLC, and compares it to the MSE. Chapter 9 describes the baseline RTC approach, while Chapter 10 describes static-analysis improvements to the RTC. Finally, Chapter 11 gives a review of related work, and Chapter 12 concludes with a summary and some directions for further research.

Each approach described in the thesis is evaluated on a number of benchmarks; a description of these benchmarks, and the inputs used in our evaluation, are given in the Appendix (Page 158). Runtime overheads are reported either as a percentage or a slowdown factor: when reported as a percentage, the overhead is computed as $\frac{t_{inst} - t_{orig}}{t_{orig}} \times 100\%$, and when reported as a slowdown factor, the overhead is computed as $\frac{t_{inst}}{t_{orig}}$, where t_{inst} is the running time of the instrumented executable, and t_{orig} is the running time of the original (uninstrumented) executable. In each case, the average reported is the arithmetic mean.

Chapter 2

Control-Transfer Attacks

This chapter describes a class of attacks called control-transfer attacks, in which an executing program is made to transfer control to an unintended, potentially arbitrary, piece of code. This is the most dangerous class of attacks, as it gives the successful attacker a lot of flexibility to access or change data on the compromised system.

Effecting such an attack requires the following two steps:

1. Supply a piece of malicious code (usually a “shellcode”, which spawns a shell through which the attacker can access the compromised system, often with increased privileges).
2. Cause the attacked program to transfer control to the shellcode.

The first step is usually easy: the shellcode can be stored in an environment variable, passed in the command-line argument, or fed as part of the input read by the program. One proposal for making this step more difficult is to have a non-executable stack space [Sol]. This approach has not been widely adopted because it limits the flexibility to apply certain programming techniques like “trampolines”, and because it can be circumvented with exploits such as “return-into-libc” (where control is transferred to the `system` library function to spawn a shell).

With an executable shellcode in place, the next step is to somehow make the program transfer control to the shellcode. Control-sensitive locations that may be overwritten to accomplish this include the following:

Return address : Probably the most commonly used approach, called “stack smashing” [Smi97], is to write the address of the shellcode into the return address field of the activation record. Then, when the current function returns, control is transferred to the shellcode.

Global offset table (GOT) : This is a table used to dynamically resolve addresses of library functions. By writing the address of the shellcode into the GOT entry for a given library function, a subsequent call to that function will transfer control to the shellcode instead.

atexit and .dtors tables : The `atexit` function allows a programmer to register functions to be called when the program terminates; the address of the registered function is stored in a statically-allocated table. The `.dtors` section in a GNU C compiled program serves a similar purpose, storing addresses of “destructor” functions that are called when the program terminates. Writing the address of the shellcode into either of these locations will cause the shellcode to be executed when the program is about to terminate.

Function pointers : If the program has an indirect function call, the attacker could overwrite the function pointer with the address of the shellcode, so a subsequent call via that function pointer will transfer control to the shellcode.

longjmp buffers : The `setjmp/longjmp` mechanism is used for non-local control transfer. A call to `setjmp` saves information about an execution context — including the program counter — in a buffer `buf` such that the call `longjmp(buf)` will transfer control to the saved location. An attacker can overwrite the program counter in `buf` to cause control to be transferred to the shellcode instead.

exec/system call arguments : C library functions like `system`, `popen`, and the `exec` family of functions execute the command specified in their arguments. An attacker could

overwrite an argument to one of these functions with a more powerful command, like `"/bin/sh"`.

Note that the return address, GOT, `atexit` table, and `.dtors` table are not part of the user-defined space, so a malicious write into these locations must violate memory safety. Function pointers, `longjmp` buffers, and library-function arguments, on the other hand, are part of the user-defined space, so they may potentially be overwritten without violating memory safety. However, their conventional usage is such that assignments to them should be very restricted, so a malicious write into one of these locations is seldom possible without an invalid write access. This observation suggests that detecting invalid write accesses is sufficient to prevent most control-transfer attacks.

The next three sections describe three known programming flaws — namely buffer overruns, format-string vulnerabilities, and erroneous `frees` — that can be exploited to write a malicious value into a control-sensitive location. In all three cases, exploiting the vulnerability requires an invalid write access into a control-sensitive location. An approach that detects invalid write accesses would detect an attempted exploit of these vulnerabilities, as well as other vulnerabilities that may not have been discovered. (The latter is an important property for preventing “day-zero” attacks.)

2.1 Buffer-Overrun Vulnerabilities

If a program writes a string into a buffer without checking against the size of the buffer, an attacker could supply a longer-than-expected string to cause the program to write beyond the end of the buffer. If the buffer is allocated on the stack, it is usually easy to exploit this flaw to overwrite the return address.

Figure 2.1(a) presents a piece of code with a buffer-overflow vulnerability. Column (b) gives the stack configuration at runtime, with the local variables `dst`, `buf`, and `fp` allocated above the return address field (`ret`) in the function’s activation record. The function `f` copies a user-supplied string (pointed to by `src`) into a local array (`buf`) until a null-character is encountered. The loop does not perform bounds-checking, so an attacker can cause a buffer

| (a) Vulnerable Code | (b) Runtime Stack | |
|--|-------------------|-----|
| <pre> void f(char *src) { 1. void (*fp)() = ... 2. char buf[16]; 3. char *dst = &buf[0]; 4. do { 5. *dst++ = *src; 6. } while (*src++ != '\0'); 7. (*fp)(); } </pre> | (buf[0]) | : |
| | | dst |
| | (buf[15]) | buf |
| | | : |
| | | fp |
| | | : |
| | | ret |
| | | src |
| | | : |
| | | |

Figure 2.1 Buffer Overrun Example.

overrun by supplying a `src` string longer than 16 bytes. The overrun can be exploited to overwrite the function pointer `fp` or the return address with the address of the attacker's code.

2.2 Format-String Vulnerabilities

The `printf` family of functions takes as input a format string and a variable number of arguments, which it outputs to a stream by interpreting %-specifiers in the format string. Figure 2.2(a) gives an example usage, along with the stack configuration during the call to `printf`.¹ The `%d` specifier causes argument 1 (the integer 1234) to be written in string representation, while the `%n` specifier writes an integer value (the output character count) into the location pointed to by argument 2 (the address of location `i`). `printf` assumes that

¹For simplicity, argument passing is assumed to occur entirely on the stack; i.e., no arguments are passed via registers.

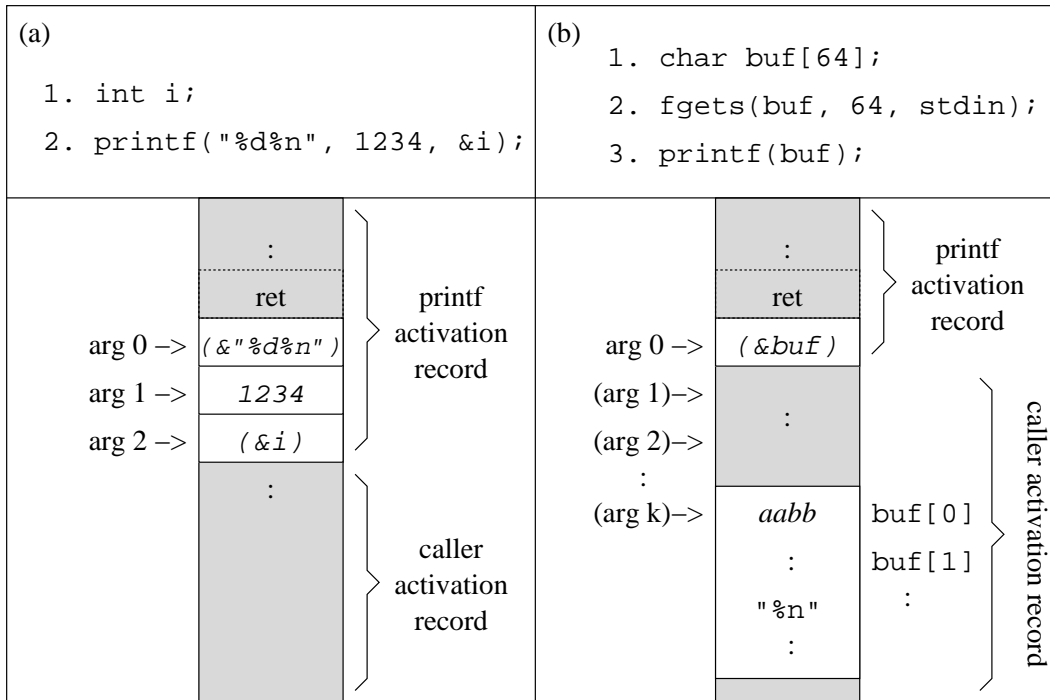


Figure 2.2 Format String Example.

the appropriate number of arguments have been pushed onto the stack, so it reads them off sequentially from its activation record as illustrated.

A format string with no %-specifiers will be output verbatim; as a result of this default behavior, programmers often call `printf` by feeding a user-supplied input string directly as the format-string argument to `printf`, as in Figure 2.2(b). Under normal usage, when the supplied input does not contain any %-specifiers, the program will behave as expected, and echo the supplied string. However, a malicious agent can supply an input string with %-specifiers to cause (essentially) any desired value to be written into any desired location in memory. If the input string contains a number k of %-specifiers, `printf` will assume that $k + 1$ arguments have been pushed onto the stack, and will interpret values on the stack as if they were supplied arguments, as shown in Figure 2.2(b). Since the `buf` array resides in the caller's activation record (as is likely to occur in a real program), a large enough k will cause `printf` to interpret the first bytes of `buf` as arguments. The attacker can thus supply the

address of a control-sensitive location in the first bytes of `buf` (the value `aabb` in the figure), followed by k %-specifiers with the last one being a `%n`. This causes the output character count (an attacker-controllable value²) to be written to the control-sensitive location whose address is `aabb`.

2.3 Erroneous Free

Heap memory management is usually implemented by storing extra information in the bytes preceding each “malloc chunk” of memory. If a program erroneously tries to free a memory block B that was not properly allocated or has already been freed, an attacker may be able to adjust the values around the beginning of B to create a bogus “malloc chunk” data structure, and trick the bookkeeping mechanism of `free` to write an arbitrary value into an arbitrary location in memory.

This section describes a specific vulnerability in the `traceroute` utility that was recently found to be exploitable [Dvo00].

2.3.1 Traceroute Vulnerability

The `traceroute` utility collects information about the path a packet takes when traveling through the internet to its destination. The `-g` option allows the user to specify up to eight gateway IP addresses, though the common usage is to specify at most one. These command-line arguments are saved using a function that `callocs` one large buffer B_i and returns sub-pieces of memory from B_i . For example, if given two gateway arguments, the function returns p_1 and p_2 pointing to sub-pieces of B_i containing the two arguments, as illustrated in Figure 2.3(a). However, after each argument is processed, the program erroneously calls `free` on each sub-piece. First, the call `free(p1)` unintentionally frees all of B_i . Next, `free(p2)` results in undefined behavior, usually causing a crash.

²Modifiers to %-specifiers allow the value of the output character count to be adjusted to arbitrary values, though in a limited range. Various tricks can be used to construct a long integer value (needed to represent the address of the shellcode); see, e.g., [scu01], for details.

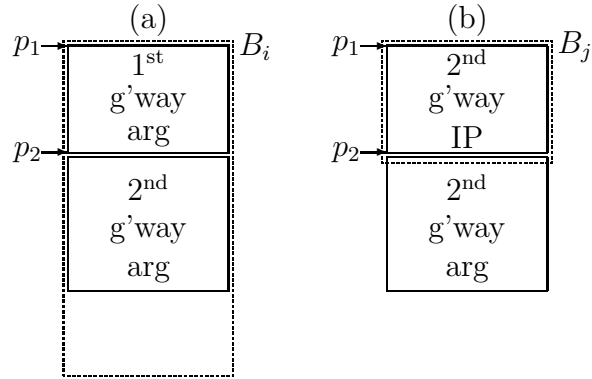


Figure 2.3

It turns out that between the calls that free p_1 and p_2 , there is a call to `calloc` that returns 16 bytes of the freed block B_i for re-use. This block (which we will call B_j) is used to store the numerical IP address of the second gateway, converted from its string representation in the second gateway argument. This means that immediately prior to the call `free(p2)`, the memory is as shown in Figure 2.3(b). Immediately after the erroneous call `free(p2)`, there is another call that frees the `calloc`'ed block B_j .

This predictable behavior gives the user control of the memory surrounding the location pointed to by p_2 . The idea behind the exploit is to adjust the values in the memory locations around p_2 to fool `free(p2)` so that it treats p_2 as if it points to the beginning of an allocated chunk. By creating a suitable memory layout to simulate internal data structures expected by `free`, the `free` function's housekeeping routines can be made to write the address of the attacker's shellcode into the Global Offset Table (GOT), which is a table used to dynamically resolve function addresses. Specifically, the exploit we tested [sor02] writes into the GOT entry for `free`, so that the final call that is supposed to free B_j actually transfers control to the shellcode.

2.3.2 GNU Memory Management

Heap memory management is implemented in the GNU C library by storing extra information in the bytes around each "malloc chunk" of memory. The layout of the extra information for an allocated and free chunk are shown in Figure 2.4. In the allocated chunk,

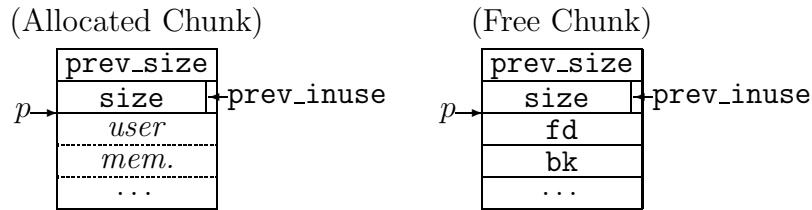


Figure 2.4

p indicates the start of the allocated user memory. For both an allocated and a free chunk, two words of extra information — containing the previous chunk size, the current chunk size, and one bit to indicate whether the previous chunk is in use — are stored immediately preceding the user memory. Free chunks of memory are stored in a doubly-linked list, with two words of the free chunk used to store pointers to the next (`fd`) and previous (`bk`) chunks in the freelist.

Consider the memory configuration of Figure 2.5. Here we have three successive memory chunks, with $chunk_2$ in use, and $chunk_3$ free (as indicated by the $prev_inuse_3$ and $prev_inuse_4$ bits). The pointers fd_3 and bk_3 maintain $chunk_3$ in the linked list of freed chunks.

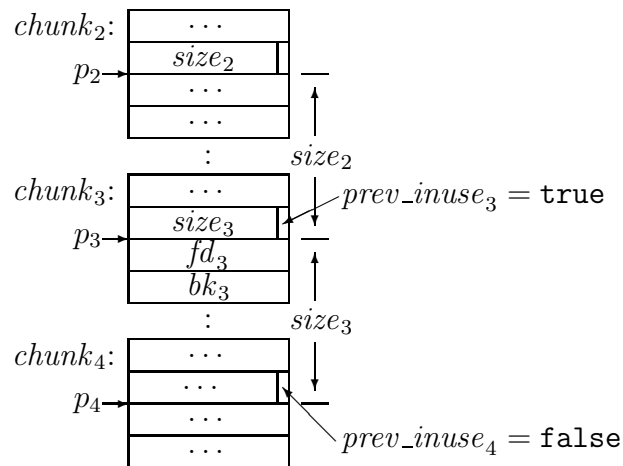


Figure 2.5

When `free(p2)` is called, the `free` function, after recognizing that `chunk3` is not in use (by checking `prev_inuse4`), will consolidate `chunk2` and `chunk3` into one big chunk. In doing so, it first unlinks `chunk3` from the linked list by executing the following instructions:

```
tmp_bk = chunk3->bk;
tmp_fd = chunk3->fd;
tmp_fd->bk = tmp_bk;
tmp_bk->fd = tmp_fd;
```

Notice that one of the effects of executing these unlinking instructions is that the location `fd3->bk` is assigned the value `bk3`. This assignment will be used by the exploit to copy the address of the shellcode into the Global Offset Table, as explained in the next section.

2.3.3 The Exploit

The goal of the exploit is to create the memory layout of Figure 2.5, where `p2` is the pointer to the second gateway argument from Figures 2.3(a) and 2.3(b), and where `chunk3` and `chunk4` are part of buffer `Bi`. This is illustrated in Figure 2.6, which shows how the memory layouts illustrated in Figures 2.3(b) and 2.5 line up.

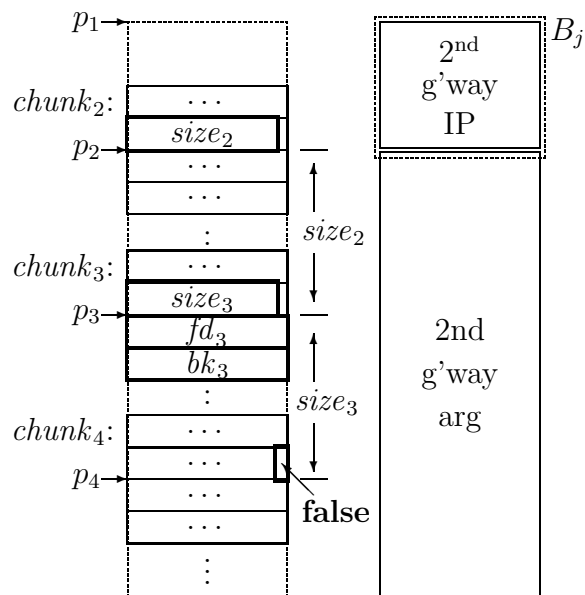


Figure 2.6

The fields that must be set by the attacker are shown in bold in Figure 2.6. The attacker can set all of those fields by choosing an appropriate string for the second gateway argument. Recall that that argument is translated to the corresponding numerical IP address, which is stored in the first 16 bytes of B_j (which includes the $size_2$ field). The method that does the translation matches the pattern `<number>.<number>.<number>.<number>`; the remainder of the argument string is ignored. Therefore, the $size_2$ field can be set by choosing an appropriate value for the fourth number, and the remainder of the argument string can be used to set the $size_3$, fd_3 , bk_3 , and $prev_inuse_4$ fields (and to contain the shellcode). The fd_3 field is set to be the address of the GOT entry for `free` minus the offset of `bk`, and the bk_3 field is set to be the address of the attacker's shellcode. Given this configuration, the unlinking done by `free` will write the address of the shellcode into the GOT entry for `free`, and the subsequent call to `free B_j` will transfer control to the shellcode.

Chapter 3

Memory-Safety Enforcer (MSE)

In the C memory abstraction, storage is described in terms of variables and heap-allocated blocks rather than registers and memory addresses. Conceptually, a program's memory space consists of a set of disjoint *objects* each with a well-defined size and lifetime. Each memory access is bound by the language semantics to a specific *intended target* object, which, roughly speaking, is the object whose address was used to compute the address of the accessed location. *Memory safety* is the condition in which every memory access falls within the bounds and lifetime of its intended target.

There are two classes of memory-safety violations: a *spatial access error* occurs when a memory access falls outside the bounds of its intended target, while a *temporal access error* occurs if the intended target of a memory access has been deallocated. In this work, the term *invalid access* will be used to refer to either class of memory-safety violation, and the terms *invalid read* and *invalid write* will refer to, respectively, a read and a write that violates memory safety.

An invalid access can give rise to unexpected behavior because it may access an arbitrary location in memory; such an access may be a security vulnerability because it may be exploited to access a specific sensitive location in memory it was not intended to access. The control-transfer attacks described in Chapter 2 all require a memory-safety violation to succeed; specifically, they require an invalid *write* into a sensitive location. Therefore,

enforcing memory safety at runtime is an effective way to protect programs from control-transfer attacks, or other less severe attacks (such as denial-of-service attacks that simply corrupt data).

This chapter describes the underlying idea behind the the Memory Safety Enforcer (MSE), an approach to detect memory safety violations for C programs at runtime. The MSE can be used as a security tool to prevent attacks from exploiting invalid accesses, and also as a debugging tool for finding errors that lead to memory-safety violations. We begin by describing a memory-safety model that satisfies the implicit requirements of the C language. We then describe the most frequently used approach to enforce memory safety, called *fat pointers*, and discuss some of its weaknesses. We then describe the MSE, which uses the *tagged memory* approach to enforce memory safety.

3.1 Memory-Safety Model

Memory accesses in C can be classified into direct and indirect accesses. Direct accesses, or accesses via directly-named non-array variables, are guaranteed by the language to be memory-safe. Indirect accesses — which we will collectively refer to by the term *dereferences* — involve either an array index ($a[i]$) or a pointer indirection ($*e$), and may be combined with a structure field or union member selector (e.g., $e.\text{mem}[i]$, $e\text{->mem}$), or be multi-level (e.g., $**e$, $e\text{->m}\text{->n}$). Indirect function calls (via a function pointer) are not considered dereferences. For simplicity of presentation, we consider only dereferences of the form $*p$ or $p\text{->mem}$ where p is a pointer variable, and assume that all dereferences in the program have been normalized to one of these forms. For example, an array index $A[i]$ can be treated as $*\text{tmp}$ where tmp has been assigned the value $A+i$, and A is treated as a pointer containing the address of the array.

Conceptually, at any given moment during program execution, each pointer value is associated with zero or one intended target. The intended target of a pointer value is the object (if any) whose address was used to compute the pointer's value. For example, after any of the assignments $p = \&x$, $p = \&x[i]$, and $p = \&x[i] + j$, the intended target of p is the

object x . When p is dereferenced, if the value of p falls outside the bounds or lifetime of x (e.g., because the values of i or j are too large, or because x has been deallocated) then a memory-safety violation occurs.

One decision that needs to be made concerns the granularity of the memory-safety model to enforce. We adopt the *outermost-object model*, in which a dereference with a given intended target is allowed to access any part of the *outermost object* containing the target. For example, after the assignment $p = \&x.m$, p 's intended target is any part of the object x (not just the m field). While this model is a good fit for the flexibility of the C language, it can give rise to some peculiarities with respect to structures.

Consider the following structure definition:

```
struct S {
    int a[10];
    int b;
} s;
```

With the outermost-object model, the dereference $s.a[i]$ would be allowed to access any part of the structure s , even if it were to overrun the bounds of the array $s.a$ and access the field $s.b$ (which would probably happen if $i = 10$).

Given this example, it may seem desirable to adopt a stricter memory-safety model that distinguishes between fields of structures, and restricts the ability of a pointer to one field to access another field. However, since the `offsetof` macro permits one to portably compute the address of one field in a structure based on another field, the stricter memory-safety model is likely to give false positives (i.e., report a deliberately designed dereference as a memory-safety violation). For example, since C requires that the address of the first field of a structure be the same as the address of the structure itself, the following two snippets (operating on the structure defined above) are equivalent in behavior:

| | |
|--|---|
| (a) | (b) |
| <pre>int * p = (int *) &s; *(p+i) = ...;</pre> | <pre>int * p = &s.a[0]; *(p+i) = ...;</pre> |

With a stricter memory-safety model, if `i` contained the value 10 (perhaps computed with the `offsetof` macro), the dereference in (a) would be permitted while the dereference in (b) would be a memory-safety violation.

Another example that is common in practice is to define a structure with an “open” array as its last field, by declaring the array to have size one:

```
struct T {
    ...
    int array[1];
} * p;
```

The structure would be allocated dynamically (with `malloc`) so that `p->array` can be treated as an array with more than one element. The stricter memory-safety model would disallow the access `p->array[i]` for $i \geq 1$.

3.2 Fat Pointers Approach

A natural approach to enforcing memory safety for pointers is to record information about the intended target with the pointer; this approach is sometimes called *fat pointers*. A fat pointer [Jim⁺02] (a.k.a., smart pointer [Ros86], safe pointer [Aus⁺94], bounded pointer [McG97], augmented pointer [Kee⁺02b], or sequential pointer [Nec⁺02]) is a triple, $\langle ptr, base, size \rangle$, where *ptr* is the pointer value, while *base* and *size* describe the intended target of the pointer. Figure 3.1 gives some example C statements and their equivalents when the pointers (`p` and `q`) are converted to fat pointers. To enforce memory safety, each dereference `*p` is checked to ensure that $p.base \leq p.ptr < p.base + p.size$.¹

There are a number of drawbacks to using fat pointers. Firstly, they do not detect temporal access errors. There are two main solutions that have been proposed to address this problem. One solution is to tag every stack frame and heap object with a unique

¹Technically, this formula is only valid if `*p` accesses one byte of memory. To handle the general case, the fat pointer triple can be defined such that the `ptr` field has the same pointer type as the original pointer, the `base` field has type `char *`, and the `size` field records the *number of bytes* in the intended target. Then, since the dereference `*p` will access `sizeof(*p)` bytes of memory, the memory-safety check of `*p` must ensure that $p.base \leq p.ptr \leq p.base + p.size - sizeof(*p.ptr)$.

| Normal Pointers | Fat Pointers |
|-------------------------------|---|
| <code>p = &x;</code> | <code>p.ptr = &x;</code> <code>p.base = &x;</code> <code>p.size = sizeof(x);</code> |
| <code>p = q+1;</code> | <code>p.ptr = q.ptr+1;</code> <code>p.base = q.base;</code> <code>p.size = q.size;</code> |
| <code>p = &a[i].m;</code> | <code>p.ptr = &a[i].m;</code> <code>p.base = &a;</code> <code>p.size = sizeof(a);</code> |
| <code>p = malloc(j);</code> | <code>p.ptr = malloc(j);</code> <code>p.base = p.ptr;</code> <code>p.size = (p.ptr==NULL)?0:j;</code> |

Figure 3.1 Fat Pointers

identifier, and to augment the fat pointer to include the identifier of its intended target, but this approach incurs a high overhead [Aus⁺94, Kee⁺02b]. Another solution is to use a garbage collector or a different memory-allocation scheme, like regions [Jim⁺02], but this solution constrains the programmer's control over memory management. Furthermore, using a garbage collector may introduce pauses that are unacceptable in certain environments (such as real-time applications), and may restrict the flexibility of the C language (e.g., CCured [Nec⁺02] does not allow pointers to stack objects to be stored in the heap).

Another problem with fat pointers is that, due to the change in pointer representation, it is difficult to interface with modules and libraries that do not use fat pointers. This is a major obstacle to their widespread use in existing systems or in legacy code. One solution is to translate fat pointers to regular pointers at function-call interfaces to modules that do not use fat pointers; a difficulty of this approach is knowing which function calls need to have their arguments translated, especially in the presence of function pointers. Another solution is to decouple the information about the intended target from the pointer itself: Jones and Kelly [Jon⁺97] store the information about the intended target in a separate data structure (a splay tree), but incur a high overhead to look up this information.

A third drawback to using fat pointers is that it is difficult to support certain unorthodox operations, or to support them efficiently. Examples of such practices are dereferencing a pointer that has been cast to an integer and back, using the difference of two pointer values to jump between objects, and declaring open arrays in structures to have size 1. Handling such idiosyncrasies with fat pointers would incur additional overhead and complication, so existing fat pointer approaches either restrict their use, do not handle them safely, or require programmer-added annotations to account for them.

3.3 Tagged Memory Approach

Instead of using fat pointers, our Memory-Safety Enforcer (MSE) adopts an alternative approach based on *tagged memory* to enforce memory safety at runtime. The underlying idea is to associate extra information with the pointed-to locations, rather than with the pointers. Specifically, each byte² of memory is tagged with one bit to indicate whether it is part of the valid target of some dereference. This bit is set to *valid* when the location is allocated, and *invalid* when it is deallocated (when exiting a scope or calling `free`). Prior to each dereference, the tag of the target location is checked: if it is tagged *invalid*, then a memory-safety violation is reported. A call to `free` is also checked to see if the target location had been previously allocated with `malloc`.³

In our implementation, the tags are maintained in a “mirror” of memory that is $\frac{1}{8}$ the size of addressable space, with a tag value of 1 representing *valid* and 0 representing *invalid*. Initially, the tags of all memory must be marked *invalid*, so the entire mirror must be zero-initialized at program startup. On systems that support demand-zero paging, the initial “allocation” and zero-initialization of a large mirror consumes negligible time and resources. The advantage of using a mirror is that looking up the tag of a byte B takes constant time, since the location of B ’s tag in the mirror can be computed directly from B ’s address. A

²We tag each *byte* of memory because C requires memory to be byte-addressable; in a language and platform with coarser addressability, we need only use one tag bit for each addressable unit of memory.

³For brevity, we use `malloc` throughout this thesis to refer to any of the heap-allocation functions: `malloc`, `calloc`, `realloc`, `valloc`, `memalign`.

disadvantage of the mirror implementation is that accessing the tags for n bytes of memory takes time linear in n . Thus, a big part of the runtime overhead is due to the setting and clearing of tags when allocating and deallocating large blocks of memory.

The tagged-memory approach can detect both spatial and temporal access errors efficiently without restricting the flexibility of C. The programmer can make arbitrary `malloc` and `free` calls, rather than be forced to use a specific memory manager or a garbage collector. The programmer is also allowed to manipulate pointers in an arbitrary fashion, including performing unorthodox pointer arithmetic or casting between pointers and integers; when such a pointer is dereferenced, the pointer value is used to look up the tag in the mirror to determine whether the access is valid.

Another benefit of the tagged-memory approach is that the tags are tamper-resistant; that is, an attacker cannot modify the tag of a given location to circumvent the MSE checks. This is because each byte of the mirror is itself tagged *invalid* within the mirror; thus, any attempt to write to the mirror (via a dereference) would be detected as an invalid write.

The tagged-memory approach also allows instrumented modules to interface cleanly with uninstrumented ones. Since there is no change in the pointer representation, there is no need to translate pointer values when calling uninstrumented functions. This is an important feature for facilitating deployment in large systems, where certain components may not have source code readily available, or may not even be written in the same language; it also allows the approach to be applied to a confined subcomponent of the system. Naturally, a memory-safety violation that occurs within an uninstrumented module would not be detected. Further, if an uninstrumented function returns a pointer to a location that is not tagged *valid*, checking a dereference of that pointer may cause a false positive to be reported. To properly account for this, a wrapper function can be written to adjust the necessary tags, or even to perform memory-safety checks. Our MSE implementation includes wrappers for standard C library functions that are vulnerable to memory-safety violations, such as `fgets`, `strcpy` and `bzero`, and functions that return pointers to objects declared within the library, such as `stat` and `ctime`.

To handle `free` calls precisely, we also maintain a hash table containing information about each `malloc`-allocated block, including its size. This allows us to check that only `malloc`-allocated blocks are freed, and tells us how many bytes of memory are being deallocated. This hash table is used for one further optimization: when dereferencing into a large block of memory (e.g., dereferencing a pointer to a structure or writing via a library function like `bzero`), if we can find that block of memory in the hash table, then the check operation associated with that dereference can take constant time instead of linear time. Note that this only improves performance when the starting address of the checked block is the starting address of a `malloc`-allocated block of memory.

The main drawback of using tagged memory is that, although it is guaranteed to detect a large class of vulnerabilities (including stack smashing), it is not guaranteed to detect all invalid accesses. However, the use of aggressive static analysis can improve the likelihood of detecting an invalid access.

3.4 MSE Classification Framework

For a given program P , we associate each program point with a unique identifier i . Let $derefs(P)$ be the set of dereferences in P , with elements of the form $\langle i, e \rangle$ to represent a dereference e at program point i . (A call to `free(e)` is treated as a dereference $*e$.) Let $locs(P)$ be the set of abstract user-defined locations in P , where each abstract location is a static representative of one or more concrete locations at runtime. We define $locs(P)$ to include the following:

- v , for each variable v (including formals) in P .
- $MALLOC_i$, for each `malloc` callsite in P : $MALLOC_i$ represents all heap objects allocated at program point i .
- $STRLIT_i$, for each string literal occurring in P : $STRLIT_i$ represents the string literal declared at program point i .

As defined, $locs(P)$ is the set of “outermost” objects declared in the program, consistent with the memory-safety model we have adopted.

The tagged-memory approach consists of two phases, *classification* and *instrumentation*:

Classification : Given a program P , compute the following two sets:

- Checked dereferences, $checked-derefs(P) \subseteq derefs(P)$, are dereferences that will be checked for memory safety at runtime; dereferences not in $checked-derefs(P)$ will not be checked.
- Tracked locations, $tracked-locs(P) \subseteq locs(P)$, are locations that will be tagged *valid* when allocated and *invalid* when deallocated at runtime. Untracked locations (locations not in $tracked-locs(P)$) will always be tagged *invalid*.

Instrumentation : Given a program P and the sets $checked-derefs(P)$ and $tracked-locs(P)$, instrument P so that the following happens at runtime:

- Initially, all locations are tagged *invalid*.
- For each tracked location that is a global variable, set its tag(s) to *valid* prior to the top-level call to `main`.
- When a tracked location is allocated on the stack or the heap, change its tag(s) from *invalid* to *valid*; a tracked static variable is marked as *valid* the first time its declaration is encountered at runtime.
- When a tracked location is deallocated (when exiting a scope or calling `free`), set its tag(s) to *invalid*.
- For each checked dereference, if any part of the target location is tagged *invalid*, report an error and halt the program.

Programs are instrumented via a C source-to-source transformation (using the Ckit front end [Ckit]). Instrumenting at the source level makes our tool portable, as an instrumented source file can be compiled on any platform that supports C.

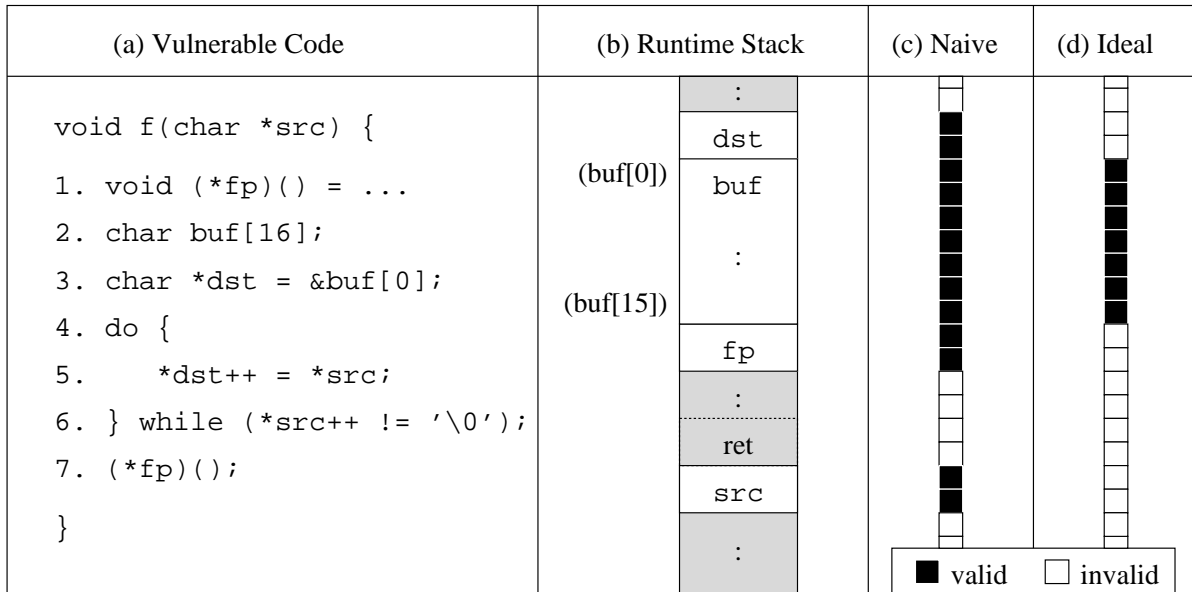


Figure 3.2 Buffer Overrun Example.

3.4.1 Naive Classification

The classification phase determines the amount of instrumentation to add. We begin by naively choosing $checked-derefs(P) = derefs(P)$ and $tracked-locs(P) = locs(P)$, so that *all* dereferences will be checked and *all* user locations will be tagged *valid* when allocated. This means that a memory-safety violation will be detected if and only if it accesses memory outside of user-defined space (such as the return address in the activation record, or memory that has been deallocated).

Figure 3.2 shows the buffer-overrun example described earlier in Section 2.1, with column (c) depicting the runtime tags associated with the naive classification scheme (where each of the user-defined locations `src`, `fp`, `buf`, and `dst` is tagged *valid*). While a stack-smashing attack that overwrites the return address (`ret`) would be detected, an attack that overwrites the function pointer `fp` would not. The goal of static analysis is to develop a better classification scheme, in particular, one that can identify $tracked-locs(P) = \{buf\}$, giving the tag layout of column (d).

3.4.2 Read-Write vs. Write-Only Checking

Let $write-derefs(P) \subseteq derefs(P)$ be the set of dereferences that are writes — i.e., that are on the left-hand-side of an assignment (or the operand of the pre- or post-fix operator $++/--$). One high-level policy we can make is to check only *writes* via dereferences, that is, to choose $checked-derefs(P) = write-derefs(P)$. As was seen in the examples in Chapter 2, control-transfer attacks usually need to exploit an invalid *write* to succeed. Since reads occur more frequently in programs than writes, checking only writes can let us gain significant improvements in performance without sacrificing too much coverage (where “coverage” means the likelihood of detecting an attack). In fact, because a smaller $checked-derefs(P)$ can lead to a smaller $tracked-locs(P)$ — as will be shown in Section 4.1 — checking only writes can improve coverage as well. However, read-write checking may still be desirable in certain high-security settings to prevent an attacker from reading confidential data, or may be useful for debugging.

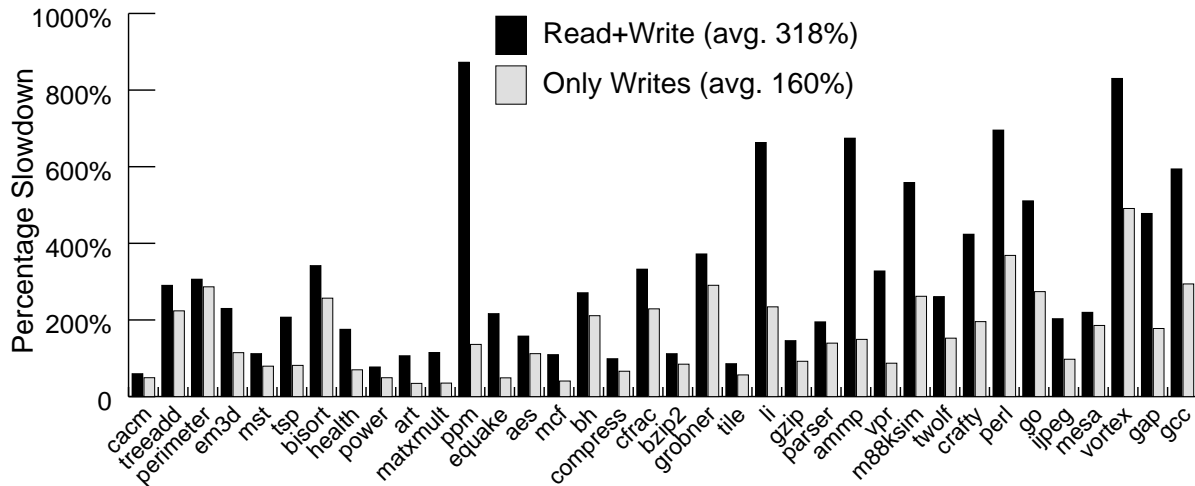


Figure 3.3 Naive Classification Runtime Overhead

Figure 3.3 gives the runtime overhead of MSE using the naive classification scheme. The benchmarks (see Appendix A for details) are sorted in increasing order of size (lines of code), to highlight any dependence on program size (in this case, the overheads are essentially

independent of program size). The percentage slowdown is computed as $\frac{t_{inst}-t_{orig}}{t_{orig}} \times 100\%$, where t_{inst} is the running time of the instrumented executable, and t_{orig} is the running time of the original (uninstrumented) executable. The average slowdown is 318% in read-write checking mode, and 160% in write-only checking mode. Not surprisingly, the overhead of write-only checking is significantly better, confirming our earlier assertion that write-only checking can result in significant gains in performance.

These runtime overheads are quite high, and are probably not acceptable for use in deployed code. They serve as a baseline case, against which improvements due to static analysis, to be presented in the following chapters, can be measured.

3.5 Improving MSE

To improve both the performance and the coverage of the Memory-Safety Enforcer, classification schemes based on several static analyses are used to identify smaller *checked-derefs* and *tracked-locs* sets. These are described in the following chapters:

- Chapter 4: Points-To Analysis
- Chapter 5: Redundant Checks Analysis
- Chapter 6: Pointer-Range Analysis

Each static analysis may be applied in either read-write checking mode or in write-only checking mode.

The static-analysis classifications are constrained to preserve coverage and soundness; that is, an invalid access that is detected by the naive classification must be reported by the improved classification, and no false positives may be reported. At a high level, the idea behind the static analyses is first to classify each dereference as either *definitely safe* or *potentially unsafe*. A dereference that is definitely safe is one that can be statically determined to never cause an invalid access, and so can be removed from *checked-derefs(P)*; only potentially unsafe dereferences will be checked. Next, a location is identified as tracked if it may be

legitimately accessed by a potentially unsafe dereference; an untracked location is one that is either never accessed by a dereference, or is only accessed by definitely safe dereferences. The goal of static analysis is to identify as few *potentially unsafe* dereferences as possible, which leads to a smaller set of *tracked* locations. Having fewer *tracked* locations (which are tagged as *valid* at runtime) increases the likelihood that a memory-safety violation will access a location tagged *invalid*, and thus be detected. Fewer potentially unsafe dereferences and tracked locations also leads to fewer runtime checks, which results in improved runtime performance.

Chapter 4

Classifications Using Points-To Analysis

Points-to analysis (specifically, the *may-points-to* flavor of points-to analysis) computes, for each pointer q and program point i , a points-to set, $pt\text{-}set_i(q)$ containing abstract locations that *may* be validly pointed to by q at program point i (just before the statement at i is executed). Figure 4.1 shows the points-to sets computed for an illustrative example: At line 1, p is assigned the address of x , so x is added to p 's points-to set for program point 2. The assignment at line 2 copies the points-to set from the right-hand-side (p) to the left-hand-side (q). As a result of the conditional at line 3, the points-to set for pp (for program point 4) states that pp may point to either p or q at runtime. For the assignment via a pointer dereference at line 4, we must use the points-to set for pp to determine which locations *may* be assigned the new value: the points-to sets computed for line 5 state that both p and q may point to either x or y .

We adopt the standard assumption that the points-to set for a pointer q includes only locations to which q may *validly* point. That is, the points-to set for q does not include

| | | | |
|---|---------------------------------|---------------------------------|----------------------------------|
| <code>int x,y,*p,*q,**pp;</code> | $pt\text{-}set_1(p) = \{\}$ | $pt\text{-}set_1(q) = \{\}$ | $pt\text{-}set_1(pp) = \{\}$ |
| <code>1. p = &x;</code> | $pt\text{-}set_2(p) = \{x\}$ | $pt\text{-}set_2(q) = \{\}$ | $pt\text{-}set_2(pp) = \{\}$ |
| <code>2. q = p+1;</code> | $pt\text{-}set_3(p) = \{x\}$ | $pt\text{-}set_3(q) = \{x\}$ | $pt\text{-}set_3(pp) = \{\}$ |
| <code>3. pp = (...) ? &p : &q;</code> | $pt\text{-}set_4(p) = \{x\}$ | $pt\text{-}set_4(q) = \{x\}$ | $pt\text{-}set_4(pp) = \{p, q\}$ |
| <code>4. *pp = &y;</code> | $pt\text{-}set_5(p) = \{x, y\}$ | $pt\text{-}set_5(q) = \{x, y\}$ | $pt\text{-}set_5(pp) = \{p, q\}$ |

Figure 4.1 Points-to Analysis Example

locations to which q may point as a result of an invalid (e.g., out-of-bounds) access. For example, after line 2, q may in fact point to y in a way that violates memory safety, but $pt\text{-}set_3(q)$ does not include y .

Precise points-to analysis is *NP*-hard [Lan⁺91, Hor97], so algorithms must trade off precision for lower complexity. *Flow-sensitive* analyses [Lan⁺92, Ema⁺94, Wil⁺95], which compute a program-point-specific points-to set for each dereference, can give precise results, but do not scale well to large programs. *Flow-insensitive* analyses [And94, Ste96, Sha⁺97, Das00], which compute for each pointer q a single points-to set $pt\text{-}set(q)$ that holds at all program points, are more efficient but less precise than flow-sensitive analysis. Our implementation uses the One-Level-Flow analysis [Das00], which is flow-insensitive and runs in near-linear time (and thus scales well to large programs).

To ensure the correctness of points-to analysis — that is, to ensure that all locations that may validly be pointed-to by q are included in $pt\text{-}set(q)$ — the whole program must be analyzed. For libraries or other components for which source code may not be available, the behavior of each function must be safely accounted for with a model of the function.

Our points-to analysis does not distinguish between fields of structures or elements of arrays. This means that if q points to some field of structure s then q 's points-to set includes all of s . It also means that if a field $s.f$ of structure s may point to some location x , then the analysis will report that *any* field of s may point to x — this simplifies the analysis but at the cost of some loss in precision. Since our analysis does not distinguish between structures, we can treat dereferences of the form $p\text{-}\>\text{mem}$ as equivalent to $*p$. Therefore, in this chapter we assume all dereferences are of the form $*p$.

4.1 Classifying Tracked Locations

For a given $checked\text{-}derefs(P)$, the corresponding $tracked\text{-}locs(P)$ must include all locations that may be a valid target of a dereference in $checked\text{-}derefs(P)$. Since the points-to set for a pointer p is guaranteed to contain any location that may be a valid target of $*p$, $tracked\text{-}locs(P)$ can be computed as the union of the points-to sets of the dereferences in

checked-derefs(P):

$$\textit{tracked-locs}(P) = \bigcup_{\langle i, *q \rangle \in \textit{checked-derefs}(P)} \textit{pt-set}_i(q)$$

This gives a concise set of locations that *may* be validly accessed by a checked dereference at runtime.

In the example from Figure 3.2, only `buf` is classified as tracked, since $\textit{pt-set}(\textit{dst}) = \{\textit{buf}\}$. The other locations, `src`, `fp`, and `dst`, are not in the points-to set of any dereferenced pointer, so they can be excluded from the *tracked-locs* set. This gives us the runtime configuration of column (d) that we wanted.

With the naive *checked-derefs*(P), we computed the corresponding *tracked-locs*(P) with this technique. Using this classification to instrument the programs, the resulting runtime performance is significantly better than the naive approach, as shown in Figure 4.2: the average slowdown is 208% for checking reads and writes (a 35% improvement over the naive classification), and is 60.3% for checking writes only (a 62% improvement over the naive classification).

4.2 Extended Points-To (EPT) Analysis

To reduce the size of *checked-derefs*(P), we use static analysis to determine whether a given pointer dereference is *definitely safe*, that is, if it can be guaranteed to never violate memory safety. A dereference that is definitely safe need not be instrumented, and thus can be removed from *checked-derefs*(P). A dereference that cannot be statically determined to be definitely safe is *potentially unsafe*, and must be included in *checked-derefs*(P).

As a first step in classifying *definitely safe* and *potentially unsafe* dereferences, we develop an Extended Points-To (EPT) analysis to conservatively identify pointers that may cause a spatial access error. In C, a pointer dereference `*p` can cause a spatial access error only if it has been assigned one of the following:

- a numeric value (including NULL);

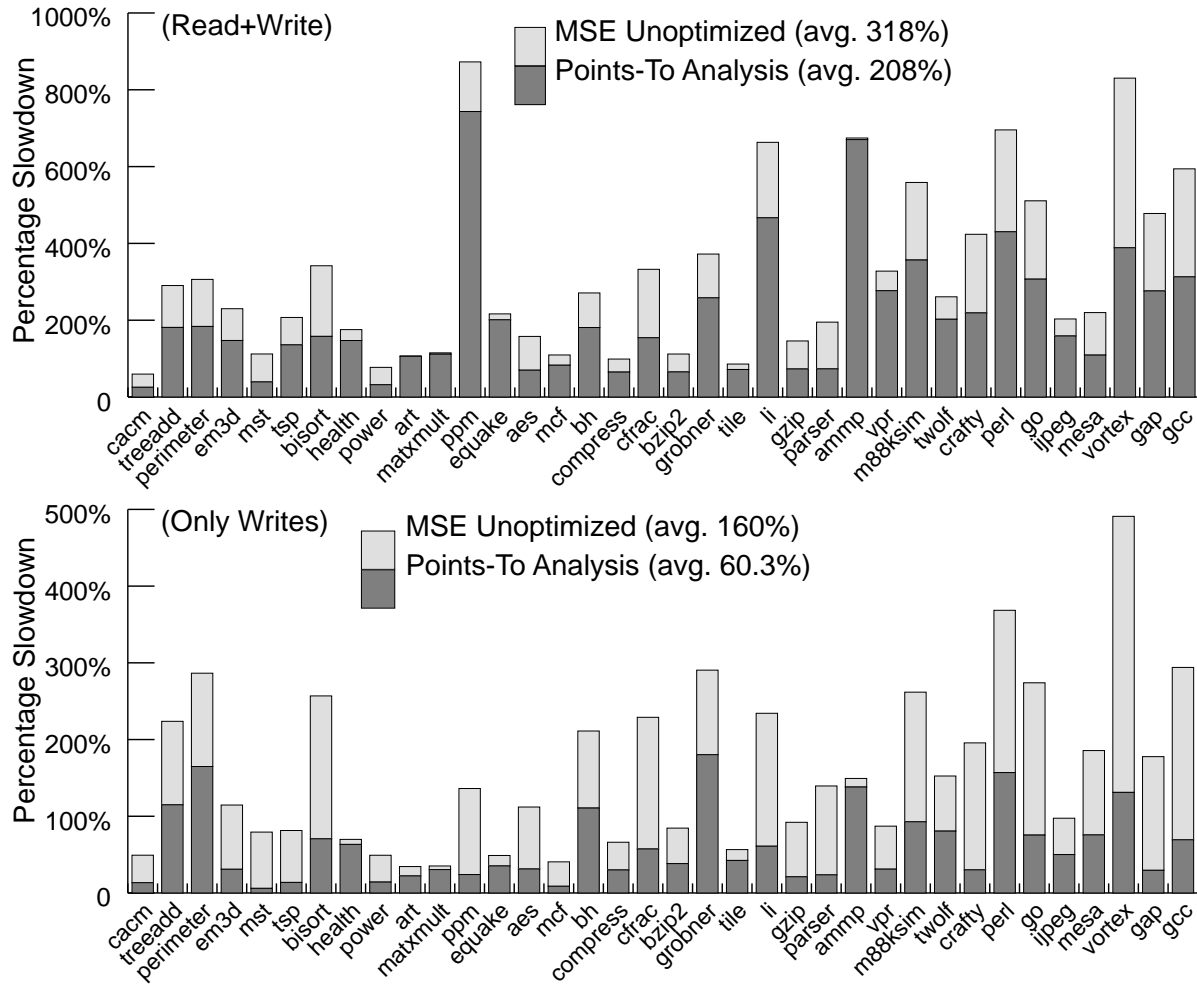


Figure 4.2 Points-To Analysis Classification: Runtime Overhead

- a computed value (a value that is the result of applying an arithmetic or bitwise operator to one or more operands);
- the address of an object that is smaller than the size of `*p`'s static type.

The extended points-to analysis introduces a special “bottom” location (\perp) to represent the target of a non-zero numeric value or a computed value. The idea is that if \perp is in the points-to set of some pointer q , then q could potentially point outside its valid target. For example, \perp is included in $pt\text{-}set(q)$ as a result of analyzing any of the following: $q = 3$, $q++$, $q = a+b$, $q = a|b$.

Note that we do not add \perp to q 's points-to set if q is assigned the value zero (`NULL`). This is because in most programming settings, dereferencing a null pointer causes a segmentation violation, which terminates the program. In the context of a security tool, a segmentation violation is a safe termination of a program that prevents an attack from successfully causing harm; therefore, we treat null-pointer dereferences as *definitely safe*. Since the `NULL` pointer (or value 0) is the default value of global variables and occurs frequently in C programs, this adjustment significantly reduces the number of potentially unsafe pointers identified.

The points-to analysis algorithm accounts for propagating the \perp location from one points-to set to another as a result of assignments. For example, assume that \perp is in p 's points-to set, and that p is in q 's points-to set. Given the assignments $a = p$ and $b = *q$, points-to analysis will determine that \perp is also in the points-to sets of both a and b .

After performing this extended points-to analysis, we classify the dereference $*p$ at program point i as potentially unsafe if any of the following holds:

1. $\perp \in pt\text{-}set_i(p)$, or
2. $x \in pt\text{-}set_i(p)$ such that $\text{sizeof}(x) \leq \text{sizeof}(*p)$, or
3. $pt\text{-}set_i(p)$ contains a stack variable, or
4. $pt\text{-}set_i(p)$ contains a heap location that may be freed.

A dereference that does not satisfy any of these criteria is definitely safe, and can be excluded from $checked\text{-}derefs(P)$.

The criteria above describe conditions under which $*p$ may cause an invalid access: 1 and 2 because p may point outside its valid target (potentially causing a spatial access error), 3 because p may point to a stack variable that is deallocated when a function returns, and 4 because p may point to a heap object that has been freed. To check criterion 4, we search the program P for calls to `free`; for each call `free(q)`, each location in q 's points-to set that is of the form `MALLOCi` is included in the set of heap locations that may be freed.

Note that since an array-index expression $A[i]$ is treated as the equivalent pointer dereference $*\mathbf{tmp}$, with $\mathbf{tmp} = A+i$, the $+$ computation adds \perp to \mathbf{tmp} 's points-to set, thus making $*\mathbf{tmp}$ a checked dereference; therefore, all array-index expressions are considered to be potentially unsafe dereferences, even if the index value is a statically known constant (an analysis for identifying array index expressions that are guaranteed to be in-bounds is described in Chapter 6: Pointer-Range Analysis).

4.2.1 May-Be-Uninitialized Pointers

Using the results of extended points-to analysis, it is possible for a dereference of an uninitialized pointer to go undetected. This possibility arises because the underlying flow-insensitive analysis assumes that all variables are properly initialized. This means that even if a pointer \mathbf{p} may be dereferenced while containing uninitialized data, $*\mathbf{p}$ may be classified as definitely safe. This is a problem, as dereferencing an uninitialized pointer may potentially be exploited by an attacker.

To prevent dereferences of uninitialized pointers from happening, we add instrumentation to zero-initialize any location that may be the source of an uninitialized value that is dereferenced. The idea is to effectively translate any potential uninitialized-pointer dereference to a null-pointer dereference, which would cause the program to crash, and thus would not be exploitable.

The set of locations that must be zero-initialized is identified by performing a simple flow-insensitive analysis. First, an assignment graph is built, with a vertex for each location in $locs(P)$, and an edge from x to y representing a possible flow of value from x to y (resulting from a direct assignment $\mathbf{y} = \mathbf{x}$ or an indirect assignment like $*\mathbf{q} = *\mathbf{p}$ where $x \in pt\text{-}set(\mathbf{p})$ and $y \in pt\text{-}set(\mathbf{q})$). Then, for each pointer in an unchecked dereference (that is, for each \mathbf{p} such that there is some $\langle i, *\mathbf{p} \rangle \notin checked\text{-}derefs(P)$), any stack or any heap location that is backward-reachable from \mathbf{p} in the assignment graph, including \mathbf{p} itself, must be zero-initialized. (Note that global or static variables are already required to be zero-initialized by the C language specification.)

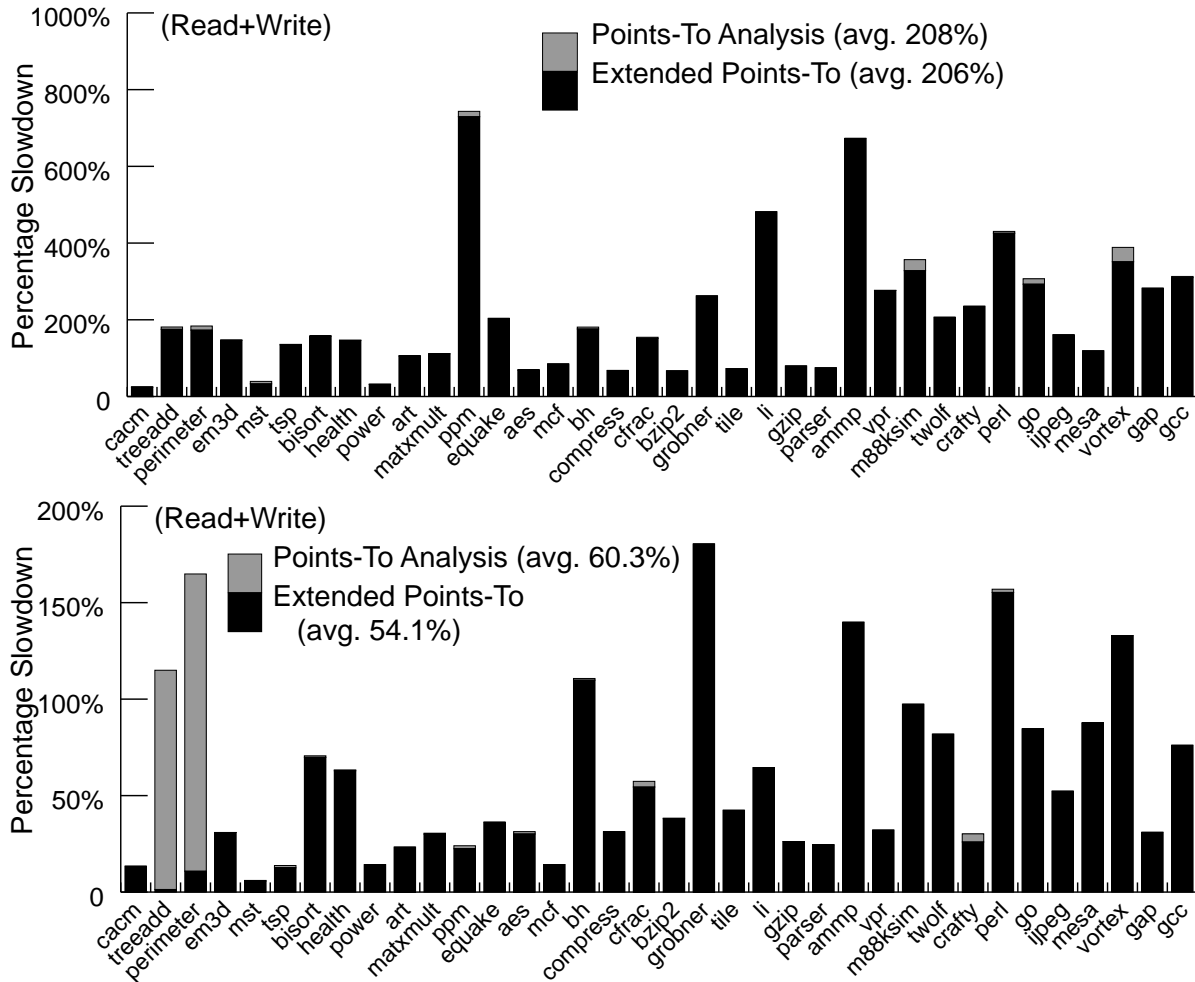


Figure 4.3 Extended Points-To Analysis: Runtime Overhead

4.2.2 Performance

Using the *checked-derefs* set consisting of potentially unsafe dereferences, and the corresponding *tracked-locs* set, the runtime overhead of the instrumented programs improved slightly overall, and significantly for two small benchmarks when checking writes only, as shown in Figure 4.3. The slowdown introduced by zero-initializing may-be-uninitialized locations (not shown in the graphs) is negligible, averaging less than 1%, with a maximum slowdown of 9% (for *mesa*).

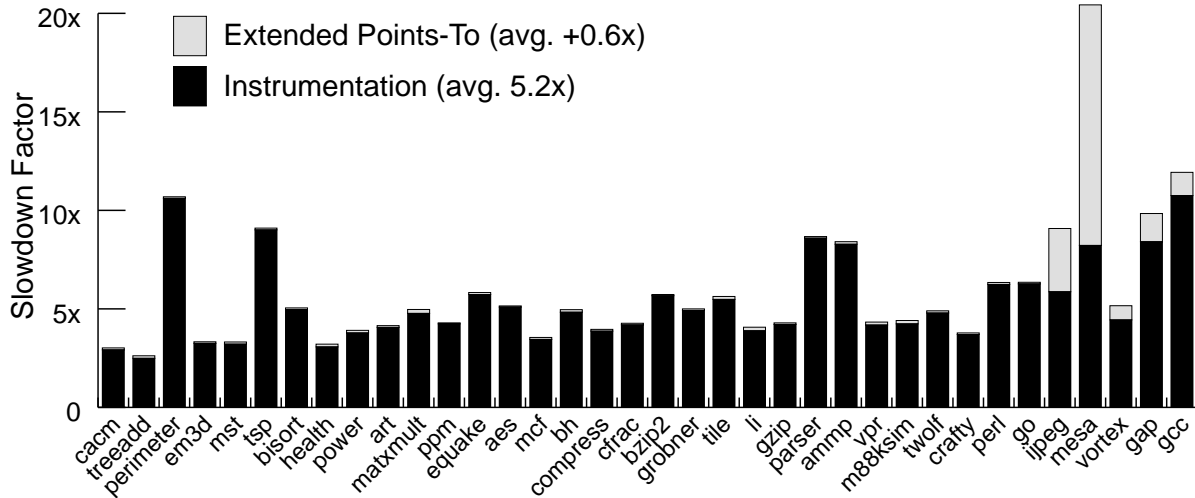


Figure 4.4 Compilation Time Slowdown

Figure 4.4 shows the compilation time slowdown, comparing the time it takes to analyze, instrument, and compile each program with the time it takes to compile the uninstrumented executable. The black portions of the bars give the slowdowns introduced by instrumentation, which are similar to the overheads for instrumenting the programs with the naive MSE classification. The lighter portions show the slowdowns due to Extended Points-To (EPT) Analysis, which is negligible in most cases, and reasonable for the larger programs.¹ This is due to EPT being based on a fast flow-insensitive points-to analysis that scales well to large programs. The analysis of `mesa` is slow because the program has some large structures (with over a thousand fields), which our implementation currently does not handle efficiently.

¹The slowdown factors are defined as follows:

Let t'_{ana} be the analysis time,

t'_{inst} be the MSE instrumentation time,

t'_{comp} be the compilation time of the instrumented source files, and

t_{comp} be the compilation time of the original (uninstrumented) source files.

The instrumentation slowdown is $\frac{t'_{inst} + t'_{comp}}{t_{comp}}$, while the analysis slowdown is $\frac{t'_{ana}}{t_{comp}}$.

4.3 Stack and Heap Locations

In the extended points-to analysis, criteria 3 and 4 (on page 40) appear to be overly conservative: $*p$ is considered potentially unsafe if p 's points-to set contains any stack variable or any heap location that is ever freed. This is to account for the possibility that the stack or heap location may be deallocated prior to the dereference, thus resulting in a dangling-pointer dereference. By using lifetime or escape analysis [Rug⁺88, Par⁺92], we could identify cases where such a dangling-pointer dereference could never occur.

To gauge the potential improvement from such an analysis, Figure 4.5 shows the runtime overhead if we modify the Extended Points-to (EPT) classification scheme to ignore criteria 3 and 4. ‘Stack OK’ assumes that stack locations are never dereferenced via dangling pointers (i.e., criterion 3 is ignored), while ‘Heap+Stack OK’ assumes that no dangling pointer is ever dereferenced (i.e., both criteria 3 and 4 are ignored). These unsafe assumptions give an upper bound on the improvement that can be attained by performing escape analysis.

The difference in performance compared to EPT is usually quite small. It may be surprising that the difference is not greater, since stack and heap locations account for most of the memory usage in these programs. The reason for the small difference in performance is that large objects are likely to be classified as *tracked* because of criterion 1, and the handling of large objects is a dominant contributor to the runtime overhead (when allocating or deallocating a large object, the tags of the entire object must be set or cleared). In the few cases where criteria 3 and 4 make a notable difference, most of the potential improvements are due to stack variables that are passed by reference to a function.

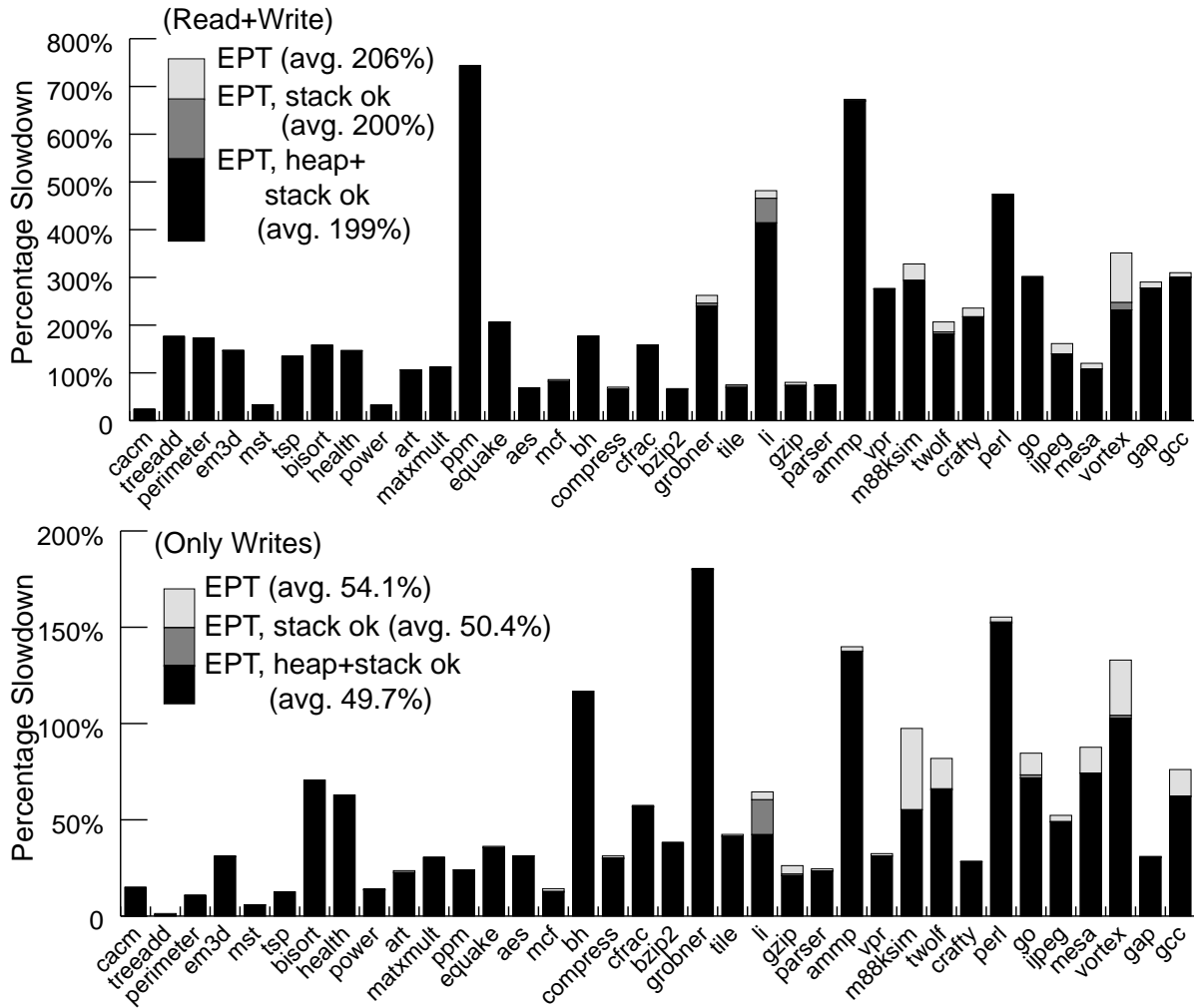


Figure 4.5 Unsafe Stack/Heap Assumptions: Runtime Overhead

Chapter 5

Redundant Checks Analysis

To reduce the overhead of MSE, we perform a dataflow analysis to detect redundant checks that can be safely eliminated. Since a runtime check of a dereference e will terminate the program if an error is detected (if the target location of e is not tagged *valid*), a subsequent check e' would never detect an error if

1. e' refers to the same target location as e , and
2. the target location has not been deallocated.

If we can statically determine that these two conditions are satisfied for the check of e' , that check is redundant and can be safely eliminated.

An intra-procedural forward dataflow analysis is performed to identify dereferences that have been checked along all paths leading to a given node:

- The elements of the underlying lattice are sets of dereference expressions.
- Two sets are computed for each CFG node n ,¹ $Checked_{in}^n$ and $Checked_{out}^n$, which contain dereference expressions for which checks would be redundant before and after n .
- For each function's enter node n_0 , $Checked_{in}^{n_0} = \emptyset$
- To safely handle function calls, two auxiliary sets are pre-computed for each function f that will be used by the analysis:

¹For convenience, we assume the control-flow graph (CFG) numbering n coincides with the program-point numbering introduced in Section 3.4.

- $MayMod(\mathbf{f})$: the set of locations that may be modified as a result of calling \mathbf{f} .
 - $MayFree(\mathbf{f})$: the set of heap locations that may be freed as a result of calling \mathbf{f} .
- The dataflow function for each node n is of the form

$$Checked_{out}^n = Checked_{in}^n \cup Gen(n) - Kill(n)$$

where $e \in Gen(n)$ if e occurs as a checked dereference in node n (i.e., if $\langle n, e \rangle \in checked-derefs(P)$) and $e \in Kill(n)$ if n may change the *lvalue* of e or free the location accessed by e (this is discussed in more detail below). Note that, unlike conventional *Gen/Kill* problems, this analysis adds the *Gen* set before removing the *Kill* set from the dataflow fact.

- The lattice meet is set intersection, since the presence of expression e in the dataflow fact means e *must* have been checked.

After performing the analysis, if node n contains a checked dereference e such that $e \in Checked_{in}^n$, then the check of e at node n would be redundant, thus $\langle n, e \rangle$ can be removed from $checked-derefs(P)$.

5.1 Affecting Locations

The *Kill* set for a given CFG node n must include all expressions e whose *lvalue* may be changed by n , or for which n may free the location accessed by e . The *lvalue* of an expression e is the *location* to which e refers, rather than the *value* stored at e (the latter is called the *rvalue* of e). For example, if \mathbf{i} contains the value 5, \mathbf{i} 's *lvalue* is the location \mathbf{i} , and its *rvalue* is 5. For a dereference expression $*\mathbf{p}$, its *lvalue* is determined by the value of the pointer \mathbf{p} . Thus, for the redundant checks analysis, $Kill(n)$ must include the dereference $*\mathbf{p}$ if n may modify the value of \mathbf{p} , or free the location accessed by $*\mathbf{p}$. Specifically, the dereference $*\mathbf{p}$ is in $Kill(n)$ if

1. n contains an assignment to \mathbf{p} , or

2. n contains an assignment to $*q$, where $p \in pt\text{-set}(q)$, or
3. n contains a call to $\text{free}(q)$, where $pt\text{-set}(p)$ intersects $pt\text{-set}(q)$, or
4. n contains a call to function f , where
 - a. $p \in \text{MayMod}(f)$, or
 - b. $pt\text{-set}(p)$ intersects $\text{MayFree}(f)$.

In conditions 1, 2, and 4a, the assignment in n may change the location to which p points, while in conditions 3 and 4b, the function call in n may deallocate the location to which p points.

The definition of *Kill* above assumes that dereferences in the program have been normalized to the form $*p$, where p is a pointer variable. There is a drawback to making this assumption: the analysis may miss identifying redundant checks for more complicated dereference expressions. Consider the following example:

| (a) | (b) |
|---------------------|---------------------------|
| 1. | 1. $\text{tmp1} = (p+1)$ |
| 2. $*(p+1) = \dots$ | 2. $*\text{tmp1} = \dots$ |
| 3. | 3. $\text{tmp2} = (p+1)$ |
| 4. $*(p+1) = \dots$ | 4. $*\text{tmp2} = \dots$ |

Program (b) is the normalized form of program (a). Note that if program (b) is analyzed, the redundant check at line 4 would not be eliminated because the dereference expression $*\text{tmp1}$ at line 2 does not match the dereference expression $*\text{tmp2}$ at line 4.

For this reason, our analysis does not, in fact, assume that dereference expressions have been normalized to the form $*p$. Instead, a dereference expression may have any form satisfying *deref-expr* in the following grammar:

$$\begin{aligned}
deref\text{-}expr & ::= *rvalue\text{-}expr \\
& \quad | \quad rvalue\text{-}expr [rvalue\text{-}expr] \\
& \quad | \quad deref\text{-}expr . member \\
& \quad | \quad rvalue\text{-}expr \rightarrow member \\
lvalue\text{-}expr & ::= *rvalue\text{-}expr \\
& \quad | \quad rvalue\text{-}expr [rvalue\text{-}expr] \\
& \quad | \quad lvalue\text{-}expr . member \\
& \quad | \quad rvalue\text{-}expr \rightarrow member \\
& \quad | \quad variable \\
rvalue\text{-}expr & ::= lvalue\text{-}expr \\
& \quad | \quad constant \\
& \quad | \quad \& lvalue\text{-}expr \\
& \quad | \quad rvalue\text{-}expr \oplus rvalue\text{-}expr
\end{aligned}$$

where *constant* is a constant (including `sizeof` expressions), *variable* is a variable identifier, *member* is a structure field or union member identifier, and \oplus is any arithmetic, logical, or boolean operator.² The grammar for *deref-expr* captures the set of side-effect free dereference expressions in C, i.e., it excludes function calls and assignment expressions (including `++` and `--`).

In the above grammar, *lvalue-expr* describes the set of expressions that have an *lvalue*; we call each such expression an *lvalue expression*.³ Note that *deref-expr* is the subset of *lvalue-expr* that excludes directly-named locations (variables, and structure or union fields of variables).

Given this more general form of dereference expressions, we need to refine the definition of the *Kill* set for the Redundant Checks Analysis. Towards this end, we define for each expression e the sets $pt\text{-}set(e)$, $alias\text{-}locs(e)$, $rvalue\text{-}affecting\text{-}locs(e)$, and $lvalue\text{-}affecting\text{-}locs(e)$

²To simplify explanation, we treat \oplus as a binary operator in this description, though it should also represent the unary arithmetic, logical, boolean, and type-cast operators, as well as the ternary `?:` operator. Extending our descriptions and rules for \oplus to the unary or ternary operators is trivial; these extensions are excluded for brevity.

³This use of the term *lvalue* may give rise to some confusion, because of the origin of the term as the “left-hand-side” of an assignment, in contrast to the term *rvalue*. Instead, we read the term *lvalue* as “location-value”, i.e., a reference to the *location* rather than the *value* of an expression, and use *rvalue* to mean the value stored at that location.

We consider an *lvalue expression* to be an expression that has an *lvalue*, not an expression that occurs in a context requiring an *lvalue*. For example, the statement `p = &i + j` contains three *lvalue* expressions, `p`, `i`, and `j`, even though `j` occurs in a context that does not require an *lvalue*.

for each expression e . These are defined formally in Figure 5.1 and described informally below:

- (a) $pt\text{-}set(e)$ extends the notion of the points-to set to arbitrary expressions. For any expression e , $pt\text{-}set(e)$ is the set of locations to which the value of e may legitimately point. For example, $pt\text{-}set(*\mathbf{p})$ includes locations that may be pointed to by locations in \mathbf{p} 's points-to set.
- (b) $alias\text{-}locs(e)$ is the set of locations to which the *lvalue* expression e may (validly) refer. In essence, for a variable \mathbf{x} , $alias\text{-}locs(\mathbf{x})$ is \mathbf{x} itself, while for a dereference $*\mathbf{p}$, $alias\text{-}locs(*\mathbf{p})$ is the points-to set of \mathbf{p} .
- (c) $rvalue\text{-}affecting\text{-}locs(e)$ is the set of locations whose value may affect the *rvalue* of expression e ; i.e., if x is in $rvalue\text{-}affecting\text{-}locs(e)$, changing the value of x may change the *rvalue* of e . For example, $rvalue\text{-}affecting\text{-}locs(*\mathbf{p}) = pt\text{-}set(\mathbf{p}) \cup \{\mathbf{p}\}$, since changing the value of \mathbf{p} or any location in \mathbf{p} 's points-to set may change the *rvalue* of $*\mathbf{p}$.
- (d) $lvalue\text{-}affecting\text{-}locs(e)$ is the set of locations whose value may affect the *lvalue* of expression e ; i.e., if x is in $lvalue\text{-}affecting\text{-}locs(e)$, changing the value of x may cause e to refer to a different location. For example, $lvalue\text{-}affecting\text{-}locs(*(\mathbf{p}+\mathbf{k})) = \{\mathbf{p}, \mathbf{k}\}$, since changing the value of \mathbf{p} or \mathbf{k} will change the location to which $*(\mathbf{p}+\mathbf{k})$ refers; and $lvalue\text{-}affecting\text{-}locs(**\mathbf{q}) = \{\mathbf{q}\} \cup pt\text{-}set(\mathbf{q})$, since changing the value of \mathbf{q} or any location in \mathbf{q} 's points-to set may change the location to which $**\mathbf{q}$ refers.

The *Kill* set for the Redundant Checks Analysis is now defined such that a dereference expression e is in $Kill(n)$ if any location in $lvalue\text{-}affecting\text{-}locs(e)$ may be assigned or freed at node n , or if any location in $alias\text{-}locs(e)$ may be freed at node n . That is, $e \in Kill(n)$ if

- n contains an assignment to e' , where $lvalue\text{-}affecting\text{-}locs(e)$ intersects $alias\text{-}locs(e')$,
or
- n contains a call to $\mathbf{free}(e')$, where $lvalue\text{-}affecting\text{-}locs(e)$ intersects $pt\text{-}set(e')$ or $alias\text{-}locs(e)$ intersects $pt\text{-}set(e')$, or

| (a) | <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 5px;"><i>exp</i></th> <th style="text-align: left; padding: 5px;"><i>pt-set(exp)</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;">CONST</td> <td style="padding: 5px;">$\{\}$</td> </tr> <tr> <td style="padding: 5px;">x</td> <td style="padding: 5px;">(from points-to analysis)</td> </tr> <tr> <td style="padding: 5px;">*e</td> <td style="padding: 5px;">$\bigcup_{p \in \text{alias-locs}(*e)} \text{pt-set}(p)$</td> </tr> <tr> <td style="padding: 5px;">$e_1 [e_2]$</td> <td style="padding: 5px;">$\bigcup_{p \in \text{alias-locs}(e_1 [e_2])} \text{pt-set}(p)$</td> </tr> <tr> <td style="padding: 5px;">$e.\text{mem}$</td> <td style="padding: 5px;">$\text{pt-set}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e \rightarrow \text{mem}$</td> <td style="padding: 5px;">$\bigcup_{p \in \text{alias-locs}(e \rightarrow \text{mem})} \text{pt-set}(p)$</td> </tr> <tr> <td style="padding: 5px;">&e</td> <td style="padding: 5px;">$\text{alias-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e_1 \oplus e_2$</td> <td style="padding: 5px;">$\text{pt-set}(e_1) \cup \text{pt-set}(e_2)$</td> </tr> </tbody> </table> | <i>exp</i> | <i>pt-set(exp)</i> | CONST | $\{\}$ | x | (from points-to analysis) | *e | $\bigcup_{p \in \text{alias-locs}(*e)} \text{pt-set}(p)$ | $e_1 [e_2]$ | $\bigcup_{p \in \text{alias-locs}(e_1 [e_2])} \text{pt-set}(p)$ | $e.\text{mem}$ | $\text{pt-set}(e)$ | $e \rightarrow \text{mem}$ | $\bigcup_{p \in \text{alias-locs}(e \rightarrow \text{mem})} \text{pt-set}(p)$ | &e | $\text{alias-locs}(e)$ | $e_1 \oplus e_2$ | $\text{pt-set}(e_1) \cup \text{pt-set}(e_2)$ |
|----------------------------|---|------------|--------------------|-------|--------|----------|---------------------------|-----------|--|-------------|---|----------------|--------------------|----------------------------|--|---------------|------------------------|------------------|--|
| <i>exp</i> | <i>pt-set(exp)</i> | | | | | | | | | | | | | | | | | | |
| CONST | $\{\}$ | | | | | | | | | | | | | | | | | | |
| x | (from points-to analysis) | | | | | | | | | | | | | | | | | | |
| *e | $\bigcup_{p \in \text{alias-locs}(*e)} \text{pt-set}(p)$ | | | | | | | | | | | | | | | | | | |
| $e_1 [e_2]$ | $\bigcup_{p \in \text{alias-locs}(e_1 [e_2])} \text{pt-set}(p)$ | | | | | | | | | | | | | | | | | | |
| $e.\text{mem}$ | $\text{pt-set}(e)$ | | | | | | | | | | | | | | | | | | |
| $e \rightarrow \text{mem}$ | $\bigcup_{p \in \text{alias-locs}(e \rightarrow \text{mem})} \text{pt-set}(p)$ | | | | | | | | | | | | | | | | | | |
| &e | $\text{alias-locs}(e)$ | | | | | | | | | | | | | | | | | | |
| $e_1 \oplus e_2$ | $\text{pt-set}(e_1) \cup \text{pt-set}(e_2)$ | | | | | | | | | | | | | | | | | | |

| (b) | <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 5px;"><i>exp</i></th> <th style="text-align: left; padding: 5px;"><i>alias-locs(exp)</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;">x</td> <td style="padding: 5px;">$\{\mathbf{x}\}$</td> </tr> <tr> <td style="padding: 5px;">*e</td> <td style="padding: 5px;">$\text{pt-set}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e_1 [e_2]$</td> <td style="padding: 5px;">$\text{pt-set}(e_1) \cup \text{pt-set}(e_2)$</td> </tr> <tr> <td style="padding: 5px;">$e.\text{mem}$</td> <td style="padding: 5px;">$\text{alias-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e \rightarrow \text{mem}$</td> <td style="padding: 5px;">$\text{pt-set}(e)$</td> </tr> </tbody> </table> | <i>exp</i> | <i>alias-locs(exp)</i> | x | $\{\mathbf{x}\}$ | *e | $\text{pt-set}(e)$ | $e_1 [e_2]$ | $\text{pt-set}(e_1) \cup \text{pt-set}(e_2)$ | $e.\text{mem}$ | $\text{alias-locs}(e)$ | $e \rightarrow \text{mem}$ | $\text{pt-set}(e)$ |
|----------------------------|---|------------|------------------------|----------|------------------|-----------|--------------------|-------------|--|----------------|------------------------|----------------------------|--------------------|
| <i>exp</i> | <i>alias-locs(exp)</i> | | | | | | | | | | | | |
| x | $\{\mathbf{x}\}$ | | | | | | | | | | | | |
| *e | $\text{pt-set}(e)$ | | | | | | | | | | | | |
| $e_1 [e_2]$ | $\text{pt-set}(e_1) \cup \text{pt-set}(e_2)$ | | | | | | | | | | | | |
| $e.\text{mem}$ | $\text{alias-locs}(e)$ | | | | | | | | | | | | |
| $e \rightarrow \text{mem}$ | $\text{pt-set}(e)$ | | | | | | | | | | | | |

| (c) | <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 5px;"><i>exp</i></th> <th style="text-align: left; padding: 5px;"><i>rvalue-affecting-locs(exp)</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;">CONST</td> <td style="padding: 5px;">$\{\}$</td> </tr> <tr> <td style="padding: 5px;">x</td> <td style="padding: 5px;">$\{\mathbf{x}\}$</td> </tr> <tr> <td style="padding: 5px;">*e</td> <td style="padding: 5px;">$\text{alias-locs}(*e) \cup \text{rvalue-affecting-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e_1 [e_2]$</td> <td style="padding: 5px;">$\text{alias-locs}(e_1 [e_2]) \cup \text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$</td> </tr> <tr> <td style="padding: 5px;">$e.\text{mem}$</td> <td style="padding: 5px;">$\text{alias-locs}(e.\text{mem}) \cup \text{rvalue-affecting-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e \rightarrow \text{mem}$</td> <td style="padding: 5px;">$\text{alias-locs}(e \rightarrow \text{mem}) \cup \text{rvalue-affecting-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">&e</td> <td style="padding: 5px;">$\text{lvalue-affecting-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e_1 \oplus e_2$</td> <td style="padding: 5px;">$\text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$</td> </tr> </tbody> </table> | <i>exp</i> | <i>rvalue-affecting-locs(exp)</i> | CONST | $\{\}$ | x | $\{\mathbf{x}\}$ | *e | $\text{alias-locs}(*e) \cup \text{rvalue-affecting-locs}(e)$ | $e_1 [e_2]$ | $\text{alias-locs}(e_1 [e_2]) \cup \text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$ | $e.\text{mem}$ | $\text{alias-locs}(e.\text{mem}) \cup \text{rvalue-affecting-locs}(e)$ | $e \rightarrow \text{mem}$ | $\text{alias-locs}(e \rightarrow \text{mem}) \cup \text{rvalue-affecting-locs}(e)$ | &e | $\text{lvalue-affecting-locs}(e)$ | $e_1 \oplus e_2$ | $\text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$ |
|----------------------------|--|------------|-----------------------------------|-------|--------|----------|------------------|-----------|--|-------------|--|----------------|--|----------------------------|--|---------------|-----------------------------------|------------------|--|
| <i>exp</i> | <i>rvalue-affecting-locs(exp)</i> | | | | | | | | | | | | | | | | | | |
| CONST | $\{\}$ | | | | | | | | | | | | | | | | | | |
| x | $\{\mathbf{x}\}$ | | | | | | | | | | | | | | | | | | |
| *e | $\text{alias-locs}(*e) \cup \text{rvalue-affecting-locs}(e)$ | | | | | | | | | | | | | | | | | | |
| $e_1 [e_2]$ | $\text{alias-locs}(e_1 [e_2]) \cup \text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$ | | | | | | | | | | | | | | | | | | |
| $e.\text{mem}$ | $\text{alias-locs}(e.\text{mem}) \cup \text{rvalue-affecting-locs}(e)$ | | | | | | | | | | | | | | | | | | |
| $e \rightarrow \text{mem}$ | $\text{alias-locs}(e \rightarrow \text{mem}) \cup \text{rvalue-affecting-locs}(e)$ | | | | | | | | | | | | | | | | | | |
| &e | $\text{lvalue-affecting-locs}(e)$ | | | | | | | | | | | | | | | | | | |
| $e_1 \oplus e_2$ | $\text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$ | | | | | | | | | | | | | | | | | | |

| (d) | <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 5px;"><i>exp</i></th> <th style="text-align: left; padding: 5px;"><i>lvalue-affecting-locs(exp)</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;">x</td> <td style="padding: 5px;">$\{\}$</td> </tr> <tr> <td style="padding: 5px;">*e</td> <td style="padding: 5px;">$\text{rvalue-affecting-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e_1 [e_2]$</td> <td style="padding: 5px;">$\text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$</td> </tr> <tr> <td style="padding: 5px;">$e.\text{mem}$</td> <td style="padding: 5px;">$\text{lvalue-affecting-locs}(e)$</td> </tr> <tr> <td style="padding: 5px;">$e \rightarrow \text{mem}$</td> <td style="padding: 5px;">$\text{rvalue-affecting-locs}(e)$</td> </tr> </tbody> </table> | <i>exp</i> | <i>lvalue-affecting-locs(exp)</i> | x | $\{\}$ | *e | $\text{rvalue-affecting-locs}(e)$ | $e_1 [e_2]$ | $\text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$ | $e.\text{mem}$ | $\text{lvalue-affecting-locs}(e)$ | $e \rightarrow \text{mem}$ | $\text{rvalue-affecting-locs}(e)$ |
|----------------------------|---|------------|-----------------------------------|----------|--------|-----------|-----------------------------------|-------------|--|----------------|-----------------------------------|----------------------------|-----------------------------------|
| <i>exp</i> | <i>lvalue-affecting-locs(exp)</i> | | | | | | | | | | | | |
| x | $\{\}$ | | | | | | | | | | | | |
| *e | $\text{rvalue-affecting-locs}(e)$ | | | | | | | | | | | | |
| $e_1 [e_2]$ | $\text{rvalue-affecting-locs}(e_1) \cup \text{rvalue-affecting-locs}(e_2)$ | | | | | | | | | | | | |
| $e.\text{mem}$ | $\text{lvalue-affecting-locs}(e)$ | | | | | | | | | | | | |
| $e \rightarrow \text{mem}$ | $\text{rvalue-affecting-locs}(e)$ | | | | | | | | | | | | |

Figure 5.1 Affecting Locations.

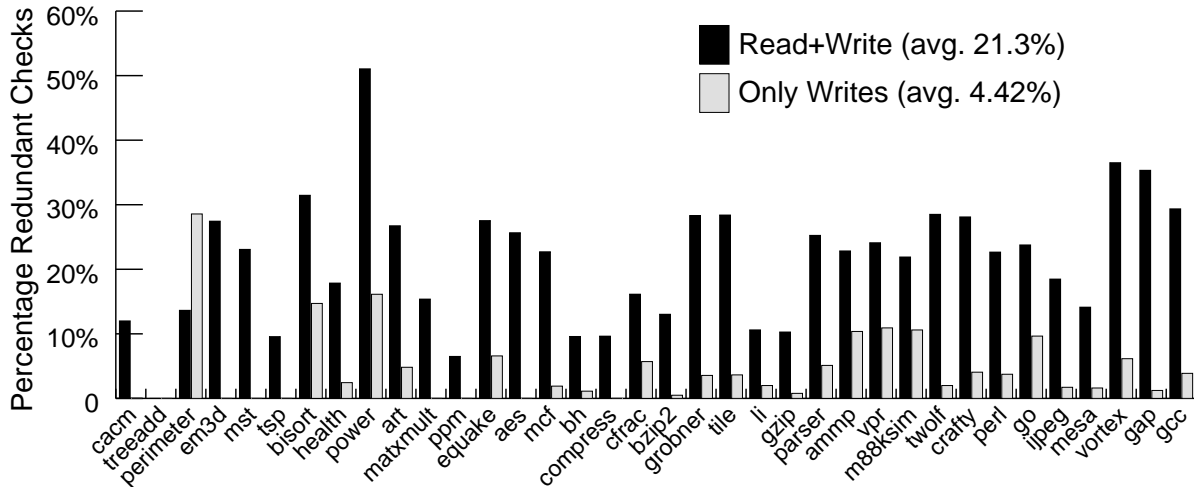


Figure 5.2 Percentage Redundant Checks

- n contains a call to function f , where $lvalue\text{-affecting}\text{-locs}(e)$ intersects $MayMod(f)$, or $lvalue\text{-affecting}\text{-locs}(e)$ intersects $MayFree(f)$, or $alias\text{-locs}(e)$ intersects $MayFree(f)$.

5.2 Evaluation

Starting with the *checked-derefs* set computed by the Extended Points-To (EPT) Analysis (described in Section 4.2), we performed Redundant Checks Analysis to remove dereferences from *checked-derefs*(P) that were found to be redundant. Figure 5.2 gives the percentage of checks that were found to be redundant in our benchmarks. On average, 21.3% of checks were found to be redundant in read-write checking mode, and 4.42% were found to be redundant in write-only checking mode. The discrepancy should not be surprising, as it is more common for programs to read from, rather than write to, the same memory location multiple times via the same dereference expression.

It is worth noting that Redundant Checks Analysis only improves *checked-derefs*(P), and does not reduce the size of *tracked-locs*(P). This is because for each redundant dereference $\langle i, *p \rangle$ eliminated from *checked-derefs*(P), there must be another $\langle j, *p \rangle$ still in *checked-derefs*(P), so the points-to set of p will still be included in *tracked-locs*(P).

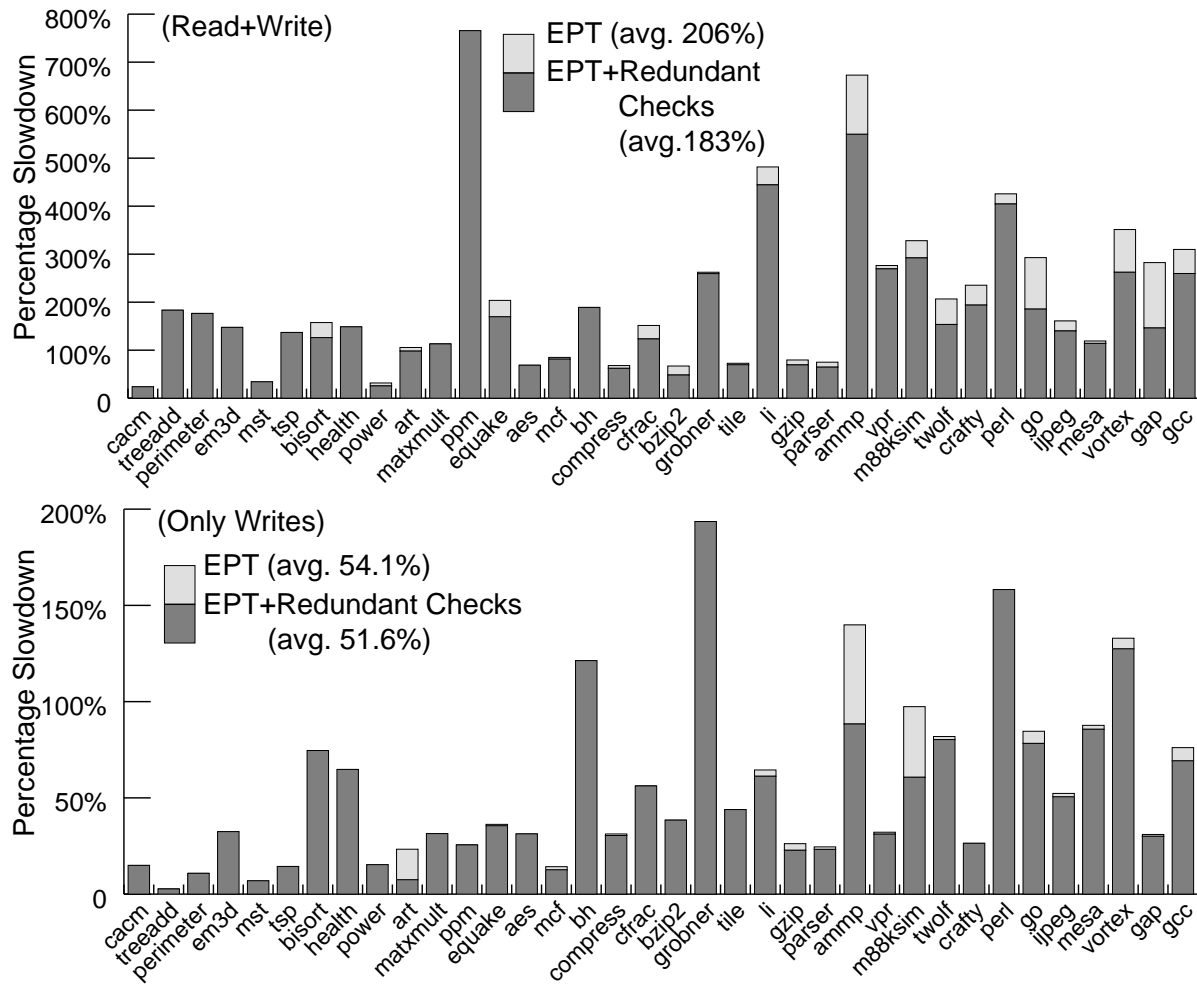


Figure 5.3 Redundant Check Analysis: Runtime Overhead

Figure 5.3 gives the runtime overhead of MSE instrumented using the Redundant Checks Analysis classification on top of EPT, compared to EPT alone. The average slowdown is 183% for checking reads and writes (improving EPT by 11%), and is 51.6% for checking writes only (improving EPT by 4.6%). Figure 5.4 gives the compilation time slowdown due to instrumentation, EPT, and Redundant Checks Analysis (as a multiple of the compilation time of the uninstrumented executable). The flow-sensitive Redundant Checks Analysis does not scale as well as the flow-insensitive EPT analysis, but the slowdown is still reasonable.

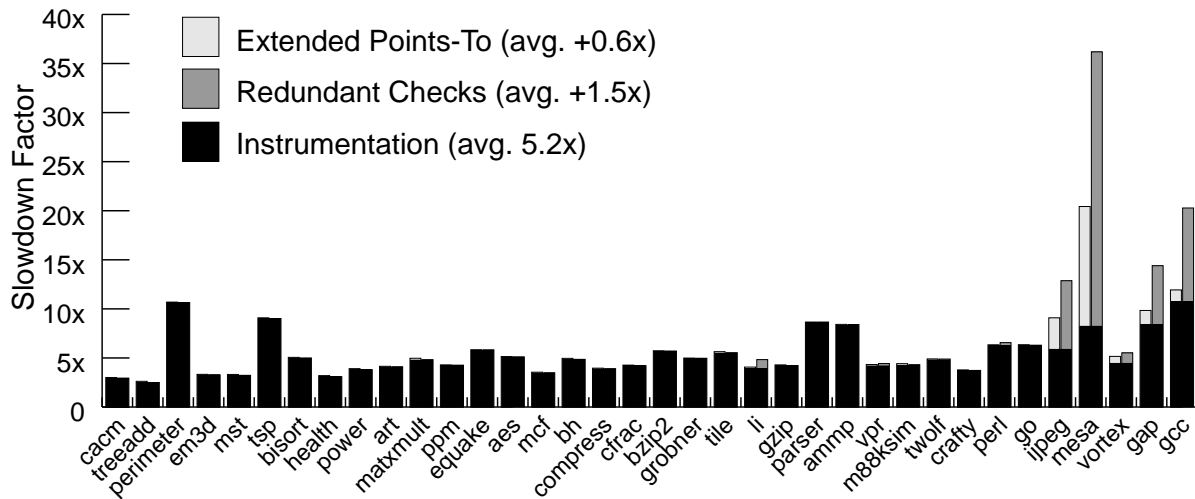


Figure 5.4 Compilation Time Slowdown

Chapter 6

Pointer-Range Analysis

Pointer-range analysis [Yon⁺04] computes the range of possible values for each pointer and array-index variable, to determine whether a given dereference is guaranteed to access in-bounds memory. A dereference that is determined to be in-bounds is *definitely safe*, and can be removed from $checked-derefs(P)$; further, if all dereferences that access a given location are found to be in-bounds, then that location may be removed from $tracked-locs(P)$.

Array-range analysis has been studied extensively in other languages (see Related Work, Section 11.3). However, the C language presents new challenges. First, arrays can be accessed indirectly via pointers, so pointer arithmetic becomes an alternative way to compute the index into an array. Second, type-casts and unions allow an array of one type to be accessed as an array of a different type, possibly with a different size. Third, even deciding what is an “array” is non-trivial, especially with heap-allocated storage, where the same mechanism (a call to `malloc`) is used whether one is allocating a single object or an array of objects. This also means that a pointer dereference to a single object and a pointer dereference to an array object cannot always be statically differentiated.

Given these features, we approach the problem of range analysis for C by treating *all* pointer dereferences as accessing an array object, and by treating each solitary object as an array with one element. Since an array-indexing expression `a[i]` is semantically equivalent to the dereference `*(a+i)`, the analysis can be described purely in terms of pointer dereferences and pointer arithmetic, rather than array accesses and array-index computation; hence the name Pointer-Range Analysis.

6.1 Representing Ranges

We define Pointer-Range Analysis as a forward dataflow-analysis problem, where at each edge of the control-flow graph (CFG), a mapping is maintained from each location x to an abstract representation of the range of values x may hold at runtime. This abstract representation must be a safe approximation (represent a superset) of the actual range of possible values. We follow the convention in dataflow analysis of performing a meet (\sqcap) at control-flow merge points, so the elements of the abstract domain must be partially ordered such that $r_1 \sqsubseteq r_2$ implies r_1 is more approximate than r_2 ; i.e., the range represented by r_1 is a superset of the range represented by r_2 .

When dealing only with numeric values, the Integer Interval Domain can be used to represent the ranges:

Integer Interval Domain $\mathcal{I} = \langle I, \sqsubseteq_i \rangle$:

- $I = \{[min, max] \mid min, max \in \mathbb{Z} \cup \{+\infty, -\infty\}, min \leq max\} \cup \{\emptyset\}$
- $[min, max]$ represents the set of integer values in the range $min \dots max$.
- $[min_1, max_1] \sqsubseteq_i [min_2, max_2]$ iff $[min_1, max_1] \supseteq [min_2, max_2]$, that is, $min_1 \leq min_2$ and $max_1 \geq max_2$.
- Note that \mathcal{I} is a lattice that satisfies our approximation requirement, with top element $\top = \emptyset$, bottom element $\perp = [-\infty, +\infty]$, and meet operator $\sqcap = \cup$.

Figure 6.1(a) demonstrates the analysis of a simple example using intervals, and shows how the computed range can be used to decide whether an array index is in bounds. When assigning a constant value to i , we can map i to a precise interval representation (e.g., $[6, 6]$ at line 4). When the two branches merge at line 8, we take the meet (union) of i 's range along the two incoming branches to get an approximate (superset) range $[3, 6]$ of possible values for i . Since this falls within the legal range of $[0, 9]$ for indexing into array a , the array index at line 9 is guaranteed to be in-bounds.

| | |
|----------------------------|--------------------|
| 1. <code>int a[10];</code> | |
| 2. <code>int i;</code> | |
| 3. <code>if(...){</code> | |
| 4. <code>i = 6;</code> | $i \mapsto [6, 6]$ |
| 5. <code>} else {</code> | |
| 6. <code>i = 3;</code> | $i \mapsto [3, 3]$ |
| 7. <code>}</code> | |
| 8. | $i \mapsto [3, 6]$ |
| 9. <code>a[i] = 0;</code> | (in-bounds) |

(a) Arrays, with Integer Intervals

| | |
|--------------------------------|---------------------------------------|
| 1. <code>int a[10];</code> | |
| 2. <code>int * p;</code> | |
| 3. <code>if(...){</code> | |
| 4. <code>p = &a[6];</code> | $p \mapsto \langle a, [6, 6] \rangle$ |
| 5. <code>} else {</code> | |
| 6. <code>p = &a[3];</code> | $p \mapsto \langle a, [3, 3] \rangle$ |
| 7. <code>}</code> | |
| 8. | $p \mapsto \langle a, [3, 6] \rangle$ |
| 9. <code>*p = 0;</code> | (in-bounds) |

(b) Pointers, with Location-Offset

Figure 6.1 In-Bounds Access Example

When dealing with pointers, however, our abstract domain must be able to capture information about a pointer’s target (the object to which the pointer may point). For a given program P , we will use the set $locs(P)$ of abstract locations (defined in Chapter 3, page 30) to represent locations to which a pointer may legally point. Each location $x \in locs(P)$ is treated as an array object, with an associated element type τ_x and element count σ_x (for solitary objects, the element count is 1). For a heap location $MALLOC_i$, it may not be possible to statically determine a precise type and count; these values are inferred from the argument to `malloc` as follows: if the argument is a constant C , we set the type to `char` and count to C ; if it is of the form $C * \text{sizeof}(\tau)$ we set the type to τ and count to C ; otherwise, we set the type to `void` and count to 0. As a shorthand, we will use the notation $|\tau|$ to represent `sizeof`(τ), the size of τ in bytes.

We now define a Location-Offset domain whose elements represent a pointer to a location plus an offset:

Location-Offset Domain $\mathcal{LO} = \langle LO, \sqsubseteq_{lo} \rangle$:

- $LO = (locs(P) \cup \{\text{NULL}\}) \times I$

- The element $\langle x, [min, max] \rangle$ represents the address of location x plus an offset in the range $[min, max]$; i.e., the range $[\&x + min \cdot |\tau_x|, \&x + max \cdot |\tau_x|]$, where τ_x is the static element type of location x .
- A NULL-targeted element $\langle \text{NULL}, [min, max] \rangle$ represents the integer range $[min, max]$.
- $\langle l_1, o_1 \rangle \sqsubseteq_{lo} \langle l_2, o_2 \rangle$ iff $l_1 = l_2$ and $o_1 \sqsubseteq_i o_2$.
- \mathcal{LO} can be converted to a lattice \mathcal{LO}^L by adding a “top” element (\top) and a “bottom” element (\perp).

Figure 6.1(b) shows a program that has the same behavior as the program in Figure 6.1(a), but which uses a pointer to indirectly access the array. At line 4, we map p to the location-offset range $\langle a, [6, 6] \rangle$ which represents the constant value $\&a + 6 \cdot |\text{int}|$. At line 8, the meet operation yields the range $[\&a + 3 \cdot |\text{int}|, \&a + 6 \cdot |\text{int}|]$ of possible values for p . At line 9, when p is dereferenced, since a has 10 elements, we can verify that the range of p falls within the legal range $[\&a + 0 \cdot |\text{int}|, \&a + 9 \cdot |\text{int}|]$, thus $*p$ will be in-bounds.

| | Location-Offset | Descriptor-Offset |
|----------------------------------|---------------------------------------|--|
| 1. <code>int a[10], b[8];</code> | | |
| 2. <code>int * p;</code> | | |
| 3. <code>if(...){</code> | | |
| 4. <code>p = &a[6];</code> | $p \mapsto \langle a, [6, 6] \rangle$ | $p \mapsto \langle a : \text{int}[10], [6, 6] \rangle$ |
| 5. <code>} else {</code> | | |
| 6. <code>p = &b[3];</code> | $p \mapsto \langle b, [3, 3] \rangle$ | $p \mapsto \langle b : \text{int}[8], [3, 3] \rangle$ |
| 7. <code>}</code> | | |
| 8. | $p \mapsto \perp$ | $p \mapsto \langle \text{UNKNOWN} : \text{int}[8], [3, 6] \rangle$ |
| 9. <code>*p = 0;</code> | (don't know) | (in-bounds) |

Figure 6.2 Multiple Target Example

The Location-Offset representation has two weaknesses. First, it can only represent a pointer to a single target location. Consider the example in Figure 6.2, where p is assigned to point to two different arrays, a and b , along the two branches. Using the location-offset

| | Location-Offset | Descriptor-Offset |
|--|--|---|
| 1. <code>int a[2];</code> | | |
| 2. <code>char *p, *q;</code> | | |
| 3. <code>p = (char *)&a[0];</code> | $p \mapsto \langle \mathbf{a}, [0, 0] \rangle$ | $p \mapsto \langle \mathbf{a} : \mathbf{int}[2], [0, 0] \rangle$ |
| 4. <code>q = p + 6;</code> | $q \mapsto \langle \mathbf{a}, [1, 2] \rangle$ | $q \mapsto \langle \mathbf{a} : \mathbf{char}[8], [6, 6] \rangle$ |
| 5. <code>*q = 0;</code> | (not in-bounds) | (in-bounds) |

Figure 6.3 Mismatched Types Example, assuming $|\mathbf{int}| = 4$.

representation, the merge point at line 8 would map p to \perp , since the elements $\langle \mathbf{a}, [6, 6] \rangle$ and $\langle \mathbf{b}, [3, 3] \rangle$ from the two incoming branches are \sqsubseteq_{lo} -incomparable, and thus the dereference at line 9 cannot be determined to be in-bounds.

The second weakness of the Location-Offset representation is that it may lose precision when handling pointer arithmetic with mismatched types. Consider the example in Figure 6.3. At line 3, p is assigned to point to an array of 2 `ints`. However, since the static type of p is `char *`, the pointer arithmetic at line 4 is `char`-based, so it must first be translated to `int`-based arithmetic before being applied to the `int`-based range $\langle \mathbf{a}, [0, 0] \rangle$. Assuming $|\mathbf{int}| = 4$ and $|\mathbf{char}| = 1$, the `char`-based addition of 6 becomes an `int`-based addition of $6 \cdot \frac{1}{4} = \frac{3}{2}$, which can only be approximated as the range $[1, 2]$. With the computed fact $\langle \mathbf{a}, [1, 2] \rangle$, the dereference at line 5 is identified as being potentially out-of-bounds, even though in fact it is definitely in-bounds.

To address these two weaknesses of the Location-Offset domain, we track the type and element count of the pointer's target separately and explicitly. First, we define the domain of Array Descriptors, whose elements describe the identity, element type, and element count of an (array) object:

Array-Descriptor Domain $\mathcal{D} = \langle D, \sqsubseteq_d \rangle$:

- $D = \text{locs}_u(P) \times T \times \mathbb{N}$

- $locs_u(P) = locs(P) \cup \{\text{UNKNOWN}\}$, where UNKNOWN represents “an unknown location”. A flat semi-lattice $\langle locs_u(P), \sqsubseteq_l \rangle$ is defined such that for all $x, y \in locs(P)$, UNKNOWN $\sqsubseteq_l x$, and $x \sqsubseteq_l y$ iff $x = y$.
 - T is the set of unqualified non-void non-array C types, with typedefs resolved to their underlying types, and with all pointer types treated as equivalent.
- The descriptor $\langle x, \tau, \sigma \rangle$ represents the location x treated as an array with at least σ elements each of type τ . For readability, we use the notation $x : \tau[\sigma]$ to represent the triple $\langle x, \tau, \sigma \rangle$. Multi-dimensional arrays are flattened; e.g., a 2×3 array of integers y is represented as $y : \text{int}[6]$.
- UNKNOWN $: \tau[\sigma]$ represents a location of unknown identity that is an array of at least σ elements of type τ .
- $(x_1 : \tau_1[\sigma_1]) \sqsubseteq_d (x_2 : \tau_2[\sigma_2])$ iff $x_1 \sqsubseteq_l x_2$ and $\tau_1 = \tau_2$ and $\sigma_1 \leq \sigma_2$.

We now define the Descriptor-Offset Domain:

Descriptor-Offset Domain $\mathcal{DO} = \langle DO, \sqsubseteq_{do} \rangle$:

- $DO = (D \cup \{\text{NULL}\}) \times I$
- The element $\langle x : \tau[\sigma], [min, max] \rangle$ represents the address of an array x with at least σ elements each of type τ , plus an offset in the range $[min \cdot |\tau|, max \cdot |\tau|]$.
- A NULL-targeted element $\langle \text{NULL}, [min, max] \rangle$ represents the integer range $[min, max]$.
- $\langle d_1, o_1 \rangle \sqsubseteq_{do} \langle d_2, o_2 \rangle$ iff $d_1 \sqsubseteq_d d_2$ and $o_1 \sqsubseteq_i o_2$.
(NULL is not \sqsubseteq_d -comparable to any member of D).
- \mathcal{DO} can be converted to a lattice \mathcal{DO}^L by adding a “top” element (\top) and a “bottom” element (\perp).

Notice that \mathcal{D} is partially ordered such that $d_1 \sqsubseteq_d d_2$ only if the size of the array described by d_1 is less than or equal to the size of the array described by d_2 . This ensures that \mathcal{DO}

satisfies the safe-approximation requirement, since if \mathbf{p} points to an array of 8 elements, it is a safe approximation to say that \mathbf{p} points to an array of 6 elements.

The rightmost columns of Figures 6.2 and 6.3 show the analysis results using Descriptor-Offset ranges. For Figure 6.2, the meet operation at line 8 sets the location component to UNKNOWN, but the type, count, and offset components are preserved: we are able to approximate the two incoming facts for \mathbf{p} by taking the smaller type-count descriptor and the superset of the interval components. When dereferencing \mathbf{p} , if \mathbf{p} maps to an element $\langle x : \tau[\sigma], [min, max] \rangle$ such that $min \geq 0$ and $max < \sigma$, then the dereference is guaranteed to be in-bounds, even if $x = \text{UNKNOWN}$ (as is the case for the dereference on line 9 of Figure 6.2).

For Figure 6.3, at line 4 we can change the type and count components of the range (assuming we know the sizes of `char` and `int`), so that array `a` is now treated as an array of 8 `chars`. This allows us to recognize that the dereference at line 5 is guaranteed to be in-bounds.

6.2 Dataflow Facts and Functions

For the dataflow analysis, each dataflow fact is a mapping $Env : Var \rightarrow \mathcal{DO}$ where Var is the set of variables in the program (Var can be extended to include fields of structure variables; this will be discussed in Section 6.6). The mapping $Env(\mathbf{x}) = r$ (sometimes written $\mathbf{x} \mapsto r$) represents the fact that variable \mathbf{x} contains a value that is safely approximated by the descriptor-offset range r . Array-typed variables with known size (i.e., with a complete type) are represented by a single representative, whose mapping safely approximates the possible values of *any* element of the array. For example, given the declaration `int a[2] = {3, 5};` the single representative `a` would be mapped to the range $\langle \text{NULL}, [3, 5] \rangle$.

Each CFG node n is associated with facts Env_{in}^n and Env_{out}^n representing the variable mappings before and after node n . Predicate nodes have two *out* facts, $Env_{out,T}^n$ and $Env_{out,F}^n$. For each node n , Env_{in}^n is the meet of the $Env_{out}^{n'}$ facts for each CFG predecessor n' of n . The

fact at the enter node of the CFG maps each global and static variable to $\langle \text{NULL}, [0, 0] \rangle$, and each automatic and register variable to \perp . Variable initializations are treated as assignments.

The dataflow transfer function for a node n defines Env_{out}^n as a function of Env_{in}^n . The dataflow transfer function that is of most interest is the one for assignments. At a node n containing the assignment $e_1 = e_2$, the right-hand-side expression e_2 is evaluated, using the mapping Env_{in}^n , to obtain a range r_2 representing the set of possible values for e_2 (the evaluation of expressions is described in Section 6.3). Env_{out}^n is defined depending on the form of e_1 :

- **Strong update:** if e_1 is a directly named variable v , then $Env_{out}^n(v) = r_2$.
- **Weak update;** if e_1 is a dereference $*p$, then for each variable v in p 's points-to set, $Env_{out}^n(v) = Env_{in}^n(v) \sqcap r_2$. That is, for any variable v that may be written by the assignment $*p = e_2$, the range of values associated with v is combined with the range of values computed for e_2 via the meet (\sqcap) operation.

The following example illustrates the difference between strong and weak updates.

| | |
|---------------|--|
| int i, *p=&i; | |
| 1. i = 5; | $Env_{out}^1(i) = \langle \text{NULL}, [5, 5] \rangle$ |
| 2. *p = 7; | $Env_{out}^2(i) = \langle \text{NULL}, [5, 7] \rangle$ |
| 3. i = 9; | $Env_{out}^3(i) = \langle \text{NULL}, [9, 9] \rangle$ |

The weak update at Line 2 approximates i 's range with $[5, 5] \sqcap [7, 7] = [5, 7]$, while the strong update at Line 3 maps i to the precise range $[9, 9]$, overwriting its previous range.

6.3 Evaluating Expressions

For a given *rvalue* expression e and mapping Env , we define the function $Eval$ to compute the *DO* range representing the set of possible values of e . Figure 6.4 gives the definition of the $Eval$ function. We assume expressions have been normalized to the forms given in Figure 6.4.

| (a) | exp | $Eval(exp, Env)$ |
|-----|-------|---|
| 1. | v | $Env(v)$ |
| 2. | $*p$ | $\prod_{x \in pt\text{-set}(p)} Env(x)$ |
| 3. | $\&v$ | $\langle v : \tau_v [\sigma_v], [0, 0] \rangle$ |
| 4. | CONST | $\langle \text{NULL}, [\text{CONST}, \text{CONST}] \rangle$ |

| (b) | exp | Static Type | $Eval(exp, Env)$ |
|-----|-------------|---|---|
| 5. | $e_1 + e_2$ | $\text{int} \times \text{int} \rightarrow \text{int}$ | $Eval(e_1, Env) +_{ii} Eval(e_2, Env)$ |
| 6. | $e_1 - e_2$ | $\text{int} \times \text{int} \rightarrow \text{int}$ | $Eval(e_1, Env) -_{ii} Eval(e_2, Env)$ |
| 7. | $e_1 + e_2$ | $\tau^* \times \text{int} \rightarrow \tau^*$ | $Eval(e_1, Env) +_{pi}^{\tau} Eval(e_2, Env)$ |
| 8. | $e_1 - e_2$ | $\tau^* \times \tau^* \rightarrow \text{int}^\dagger$ | $Eval(e_1, Env) -_{pp}^{\tau} Eval(e_2, Env)$ |

[†] The result of subtracting two pointers actually has an implementation-defined type `ptrdiff_t`.

| (c) | exp | $Eval(exp, Env)$ |
|-----|------------------|--|
| 9. | $e_1 * e_2$ | $Eval(e_1, Env) \text{ times } Eval(e_2, Env)$ |
| 10. | e_1 / e_2 | $Eval(e_1, Env) \text{ div } Eval(e_2, Env)$ |
| 11. | $e_1 \% e_2$ | $Eval(e_1, Env) \text{ mod } Eval(e_2, Env)$ |
| 12. | $e_1 \oplus e_2$ | \perp |

Figure 6.4 Evaluating expressions

For a variable v , its range is looked up in the mapping Env . For a dereference $*p$, the ranges for all the variables in p 's points-to set, as obtained from Env , are combined with the meet operator. The address-of expression $\&v$ evaluates to a *DO* range representing a pointer to v with offset 0. An integer constant evaluates to a *DO* range representing that constant.

The additive operators $+$ and $-$ are classified into four classes based on their static type signatures, as shown in Figure 6.4 (b).¹ The semantics of these four classes of operators, in terms of integer arithmetic, are given in Figure 6.5. Note that the semantics of the pointer arithmetic operators include an implicit multiplication by the size of τ . For each class, a corresponding operator (one of $+_{ii}$, $-_{ii}$, $+_{pi}^{\tau}$, $-_{pp}^{\tau}$) is defined that performs the corresponding

¹ *int+pointer* addition is assumed to have been converted to *pointer+int* by flipping its operands; *pointer-int* can be converted to *pointer+(0-int)*; note also that *pointer+pointer* and *int-pointer* are not defined in C.

| Operator : Type | Integer Semantics |
|---|-------------------------------------|
| $+$: $\text{int} \times \text{int} \rightarrow \text{int}$ | $i_1+i_2 \equiv i_1 + i_2$ |
| $-$: $\text{int} \times \text{int} \rightarrow \text{int}$ | $i_1-i_2 \equiv i_1 - i_2$ |
| $+$: $\tau^* \times \text{int} \rightarrow \tau^*$ | $p+i \equiv p + (i \cdot \tau)$ |
| $-$: $\tau^* \times \tau^* \rightarrow \text{int}$ | $p_1-p_2 \equiv (p_1 - p_2)/ \tau $ |

Figure 6.5 C Addition and Subtraction.

operation on ranges in the DO domain. These range operators are of particular interest to pointer-range analysis, and are described in Sections 6.3.1 and 6.3.2.

The three integral operators $*$, $/$, and $\%$ also have corresponding operators (*times*, *div*, *mod*) defined on the DO domain. Each of these operators computes an approximation of the appropriate arithmetic operation if its two operands are integer ranges (i.e., have a `NULL` target); if any of its operands has a non-`NULL` target, the expression evaluates to \perp .

Expressions that use an operator not covered by lines 5–11 of Figure 6.4 are evaluated to \perp .

6.3.1 Well-Typed Arithmetic

An arithmetic operation is well typed if the actual types of the arguments match the types expected by the operation. With the descriptor-offset domain, a targeted range $\langle x : \tau[\sigma], o \rangle$ represents a value of type τ^* , while a `NULL`-targeted range $\langle \text{NULL}, o \rangle$ represents a value of type `int`.

The addition and subtraction of two integer intervals can be safely approximated by the following equations:²

- $[min_1, max_1] + [min_2, max_2] = [min_1 + min_2, max_1 + max_2]$
- $[min_1, max_1] - [min_2, max_2] = [min_1 - max_2, max_1 - min_2]$

²For brevity, we omit details concerning infinite bounds, which are handled by setting respectively the upper/lower bound to plus/minus infinity if either argument needed to compute the bound is infinite.

Well-typed arithmetic on descriptor-offset ranges can be evaluated by applying these equations to the interval components of the ranges:

- Integer addition ($+_{ii}$) of two NULL-targeted ranges:

$$\begin{aligned} & \langle \text{NULL}, [min_1, max_1] \rangle +_{ii} \langle \text{NULL}, [min_2, max_2] \rangle \\ &= \langle \text{NULL}, [min_1 + min_2, max_1 + max_2] \rangle \end{aligned}$$

- Integer subtraction ($-_{ii}$) of two NULL-targeted ranges:

$$\begin{aligned} & \langle \text{NULL}, [min_1, max_1] \rangle -_{ii} \langle \text{NULL}, [min_2, max_2] \rangle \\ &= \langle \text{NULL}, [min_1 - max_2, max_1 - min_2] \rangle \end{aligned}$$

- Pointer addition ($+_{pi}^\tau$) of a τ -based range and a NULL-targeted range:

$$\begin{aligned} & \langle x : \tau[\sigma], [min_1, max_1] \rangle +_{pi}^\tau \langle \text{NULL}, [min_2, max_2] \rangle \\ &= \langle x : \tau[\sigma], [min_1 + min_2, max_1 + max_2] \rangle \end{aligned}$$

- Pointer subtraction ($-_{pp}^\tau$) of two ranges with the same target location

$x \neq \text{UNKNOWN}$ and the same element type τ :

$$\begin{aligned} & \langle x : \tau[\sigma], [min_1, max_1] \rangle -_{pp}^\tau \langle x : \tau[\sigma], [min_2, max_2] \rangle \\ &= \langle \text{NULL}, [min_1 - max_2, max_1 - min_2] \rangle \end{aligned}$$

Note that pointer subtraction requires that the two target locations x refer to the same location. Pointer subtraction of two ranges with possibly different or UNKNOWN target locations evaluates to \perp .

6.3.2 Mismatched-Type Arithmetic

An arithmetic operation that is not well typed can arise because C permits casting between pointers to different types, and between integers and pointers; it can also arise from the use of unions. This section addresses the handling of arithmetic operations on ranges with mismatched types. This includes integer addition ($+_{ii}$) with a pointer-typed argument, and pointer addition or subtraction where the type of the operation does not match the argument type.

The memory-safety model adopted by the MSE (described in Section 3.1) allows a pointer with an intended target x to access *any* component of x . With this model, we can weaken the definition of the array-descriptor ordering \sqsubseteq_d defined on page 60 so that:

- $(x_1 : \tau_1[\sigma_1]) \sqsubseteq_d (x_2 : \tau_2[\sigma_2])$ iff $x_1 \sqsubseteq_l x_2$ and $|\tau_1[\sigma_1]| \leq |\tau_2[\sigma_2]|$.

That is, d_1 is a safe approximation of d_2 if the array described by d_1 is smaller than the array described by d_2 , regardless of the element types of the descriptors.

This means that if the size of each type is known at analysis time, we can convert a range's type from τ_a to τ_b as follows:

$$\langle x : \tau_a[\sigma], [min, max] \rangle \Longrightarrow \langle x : \tau_b \left[\left\lfloor \sigma \cdot \frac{|\tau_a|}{|\tau_b|} \right\rfloor \right], \left[\left\lfloor min \cdot \frac{|\tau_a|}{|\tau_b|} \right\rfloor, \left\lceil max \cdot \frac{|\tau_a|}{|\tau_b|} \right\rceil \right] \rangle \quad (6.1)$$

We can also transform the base type of a pointer addition by adjusting the right-hand-side interval. A τ_b -based pointer addition ($+_{pi}^{\tau_b}$), where the right-hand-side is NULL-targeted, can be converted to a τ_a -based addition as follows:

$$r_1 +_{pi}^{\tau_b} \langle \text{NULL}, [min_2, max_2] \rangle \Longrightarrow r_1 +_{pi}^{\tau_a} \langle \text{NULL}, \left[\left\lfloor min_2 \cdot \frac{|\tau_b|}{|\tau_a|} \right\rfloor, \left\lceil max_2 \cdot \frac{|\tau_b|}{|\tau_a|} \right\rceil \right] \rangle \quad (6.2)$$

Transformation (6.1) or (6.2) can be used to eliminate any type mismatch, to get a well-typed operation that can be evaluated by the equations in Section 6.3.1.

Revisiting the Figure 6.3 example, the addition $\mathbf{p} + \mathbf{6}$ at line 4 has a type mismatch, because \mathbf{p} maps to an `int`-based range, while the addition is `char`-based. We can apply either transformation (6.1) or (6.2), to get the following results:

$$\begin{aligned} & \langle a : \mathbf{int}[2], [0, 0] \rangle +_{pi}^{\mathbf{char}} \langle \text{NULL}, [6, 6] \rangle \\ &= (6.1) \Rightarrow \langle a : \mathbf{char}[8], [0, 0] \rangle +_{pi}^{\mathbf{char}} \langle \text{NULL}, [6, 6] \rangle = \langle a : \mathbf{char}[8], [6, 6] \rangle \\ &= (6.2) \Rightarrow \langle a : \mathbf{int}[2], [0, 0] \rangle +_{pi}^{\mathbf{int}} \langle \text{NULL}, [1, 2] \rangle = \langle a : \mathbf{int}[2], [1, 2] \rangle \end{aligned}$$

Because of the floor and ceiling operations, there may be some loss in precision as a result of applying either transformation (6.1) or (6.2). It is therefore important to choose a transformation that minimizes loss of precision. In practice, the size of one of the types τ_a, τ_b is usually a multiple of the size of the other (making either $\frac{|\tau_a|}{|\tau_b|}$ or $\frac{|\tau_b|}{|\tau_a|}$ a whole number), so that at least one of the transformations will result in no loss of precision.

Transformations (6.1) and (6.2) can only be applied if the sizes of types are known at analysis time. Since the MSE is designed to be portable across all platforms, specific sizes of types cannot be assumed. However, some safe approximations can still be made to get results that are more precise than \perp , by making use of portable information about the sizes of types as defined or implied in the C language specification:

1. $|\mathbf{char}| = 1$
2. $|\mathbf{char}| \leq |\tau|$ for any non-void C type τ .
3. $|\mathbf{char}| \leq |\mathbf{short}| \leq |\mathbf{int}| \leq |\mathbf{long}| \leq |\mathbf{long\ long}|$
4. $|\mathbf{float}| \leq |\mathbf{double}| \leq |\mathbf{long\ double}|$
5. $|\tau[\sigma]| = |\tau| \cdot \sigma$
6. $|\mathbf{union}\ \{\tau_1 \dots \tau_n\}| \geq \max_{i=1\dots n}(|\tau_i|)$
7. $|\mathbf{struct}\ \{\tau_1 \dots \tau_n\}| \geq \sum_{i=1}^n |\tau_i|$
8. $|\mathbf{struct}\ \{\tau_1 \dots \tau_n\}| \leq |\mathbf{struct}\ \{\tau_1 \dots \tau_n \dots\}|$
9. $|\tau_1*| = |\tau_2*|$ for any C types τ_1, τ_2 .

Item 1 implies that `char`-pointer arithmetic is equivalent to integer arithmetic ($+\overset{\mathbf{char}}{pp} \equiv +_{ii}$, $-\overset{\mathbf{char}}{pp} \equiv -_{ii}$). Item 6 states that a union type is at least as large as its largest member, while item 7 states that a struct type is at least as large as the sum of its constituents' sizes (it may be larger due to padding). Item 8 takes advantage of a subtype relationship between two structures that share a common initial sequence. Item 9, which states that all pointers are of the same size, is strictly speaking an unsafe assumption, but it is all but implied by the requirements that all pointers can be cast to `void *` without loss of information, and that the return value of `malloc` can be safely cast to any pointer type. We therefore assume it to be true.

The first safe approximation, which arises often because of the way we normalize multi-dimensional arrays, is to convert a $\tau[\sigma]$ -based pointer addition, where $\tau[\sigma]$ is an array type, to a τ -based pointer addition. This is done by applying transformation (6.2) with the knowledge that $\frac{|\tau[\sigma]|}{|\tau|} = \sigma$:

$$r_1 +_{pi}^{\tau[\sigma]} \langle \text{NULL}, [min_2, max_2] \rangle \implies r_1 +_{pi}^{\tau} \langle \text{NULL}, [min_2 \cdot \sigma, max_2 \cdot \sigma] \rangle$$

Next, if we only know the relative sizes of two types, we can make the following approximations for transformation (6.2).

$$\begin{aligned} \text{If } |\tau_b| \leq |\tau_a|, \quad & r_1 +_{pi}^{\tau_b} \langle \text{NULL}, [min_2, max_2] \rangle \implies \\ & r_1 +_{pi}^{\tau_a} \langle \text{NULL}, \left[\left(\begin{array}{cc} min_2 & \text{if } min_2 \leq 0 \\ 0 & \text{otherwise} \end{array} \right), \left(\begin{array}{cc} max_2 & \text{if } max_2 \geq 0 \\ 0 & \text{otherwise} \end{array} \right) \right] \rangle \quad (6.2a) \end{aligned}$$

$$\begin{aligned} \text{If } |\tau_a| \leq |\tau_b|, \quad & r_1 +_{pi}^{\tau_b} \langle \text{NULL}, [min_2, max_2] \rangle \implies \\ & r_1 +_{pi}^{\tau_a} \langle \text{NULL}, \left[\left(\begin{array}{cc} min_2 & \text{if } min_2 \geq 0 \\ -\infty & \text{otherwise} \end{array} \right), \left(\begin{array}{cc} max_2 & \text{if } max_2 \leq 0 \\ +\infty & \text{otherwise} \end{array} \right) \right] \rangle \quad (6.2b) \end{aligned}$$

For the pointer addition $p + 6$ at line 4 of Figure 6.3, since we know $|\text{char}| \leq |\text{int}|$, we can apply transformation (6.2a) to get:

$$\begin{aligned} \langle a : \text{int}[2], [0, 0] \rangle +_{pi}^{\text{char}} \langle \text{NULL}, [6, 6] \rangle & \implies \langle a : \text{int}[2], [0, 0] \rangle +_{pi}^{\text{int}} \langle \text{NULL}, [0, 6] \rangle \\ & = \langle a : \text{int}[2], [0, 6] \rangle \end{aligned}$$

Note that the resulting range is a safe approximation (superset) of the more precise range $\langle a : \text{int}[2], [1, 2] \rangle$ obtained earlier with exact size information. A similar approximation can be made for transformation (6.1), but only in one direction:

$$\begin{aligned} \text{If } |\tau_b| \leq |\tau_a|, \text{ let } n \text{ be such that } 1 \leq n \leq \frac{|\tau_a|}{|\tau_b|}, \\ \text{then } \langle x : \tau_a[\sigma], [min, max] \rangle & \implies \\ \langle x : \tau_b[n \cdot \sigma], \left[\left(\begin{array}{cc} min & \text{if } min \geq 0 \\ -\infty & \text{otherwise} \end{array} \right), \left(\begin{array}{cc} max & \text{if } max \leq 0 \\ +\infty & \text{otherwise} \end{array} \right) \right] \rangle & \quad (6.1a) \end{aligned}$$

A key here is that $|\tau_a[\sigma]| \geq |\tau_b[n \cdot \sigma]|$, which ensures that the right-hand-side of the transformation is a safe approximation of the left-hand-side. If τ_1 and τ_2 are scalar types, the exact ratio $\frac{|\tau_a|}{|\tau_b|}$ is not portably defined, so the only safe value for n is 1. But if τ_a is an aggregate type, a safe n can be obtained by counting the number of elements in τ_a that are

at least as big as τ_b . For example, $\frac{|struct\{int\ [2],\ long,\ char\}|}{|int|} \geq 3$. It is then safe to multiply the σ component of the resultant range by n .

Thus, when evaluating the pointer addition

$$\langle x : \tau_a[\sigma], o_1 \rangle +_{pi}^{\tau_b} \langle \text{NULL}, o_2 \rangle$$

if $|\tau_a| \leq |\tau_b|$, only transformation (6.2b) can be applied. But if $|\tau_b| \leq |\tau_a|$, there is a choice between (6.1a) and (6.2a). As was the case for transformations (6.1) and (6.2), it is important to choose the transformation that minimizes the loss of precision. In general, transformation (6.1a) is more precise if the left-hand-side offset o_1 is $[0, 0]$; otherwise (6.2a) is more precise.

6.4 Predicates

For predicates at branch nodes, we can often improve the ranges for the predicate variables along the two branches. For example, along the true outgoing edge from the predicate $v_1 == v_2$, we know that v_1 and v_2 must contain the same value or point to the same location; thus, any part of v_1 's range that is not part of v_2 's range can be discarded, and vice versa. Similarly, along the true edge out of $v_1 <= v_2$, we can discard any part of v_1 's range that is strictly greater than all values in v_2 's range, and we can discard any part of v_2 's range that is strictly less than all values in v_1 's range.

To make these ideas precise, we define an *intersect* function that computes a safe approximation of the intersection of two *DO* ranges:

1. $intersect(\langle \text{NULL}, o_1 \rangle, \langle \text{NULL}, o_2 \rangle) = \langle \text{NULL}, o_1 \cap o_2 \rangle$
2. If $x \neq \text{UNKNOWN}$, then

$$intersect(\langle x : \tau[\sigma_1], o_1 \rangle, \langle x : \tau[\sigma_2], o_2 \rangle) = \langle x : \tau[\max(\sigma_1, \sigma_2)], o_1 \cap o_2 \rangle$$
3. If $x_1 \neq x_2$ and $x_1, x_2 \neq \text{UNKNOWN}$ and $o_1 \subseteq [0, \sigma_1 - 1]$ and $o_2 \subseteq [0, \sigma_2 - 1]$, then,

$$intersect(\langle x_1 : \tau_1[\sigma_1], o_1 \rangle, \langle x_2 : \tau_2[\sigma_2], o_2 \rangle) = \top$$
4. Otherwise, $intersect(\langle d_1, o_1 \rangle, \langle d_2, o_2 \rangle) = \langle d_1, o_1 \rangle$

For two integer (NULL-targeted) ranges, the intersection, given by rule 1, is straightforward. Rule 2 applies to two ranges with the same target location and element type, and takes the intersection of the offset components; if $\sigma_1 \neq \sigma_2$, the smaller σ could only have arisen due to an imprecision in applying a transformation, so we can safely restore the larger σ . Rule 3 recognizes that if two ranges refer to two different locations, then they do not intersect. The “in-bounds” conditions are necessary because an out-of-bounds offset from x_1 could potentially refer to memory in x_2 and vice versa. Finally, if none of these rules apply — that is, if at least one range has an UNKNOWN target, or if at least one range is not in-bounds, or if one range is NULL-targeted and the other is not — then rule 4 uses the left-hand-side range as a safe approximation of the intersection of the two ranges.

For an equality comparison $v_1 == v_2$ at node n , the mappings along the true outgoing edge, $Env_{out,T}^n(v_1)$ and $Env_{out,T}^n(v_2)$, would be computed as follows:

$$\begin{aligned} Env_{out,T}^n(v_1) &= \text{intersect}(Env_{in}^n(v_1), Env_{in}^n(v_2)) \\ Env_{out,T}^n(v_2) &= \text{intersect}(Env_{in}^n(v_2), Env_{in}^n(v_1)) \end{aligned}$$

For a relational comparison $v_1 <= v_2$ at node n , we want to compare v_1 's range against only the *max* component of v_2 , and compare v_2 's range against only the *min* component of v_1 . We define the functions *open-min* and *open-max*:

$$\begin{aligned} \text{open-min}(\langle d, [min, max] \rangle) &= \langle d, [-\infty, max] \rangle \\ \text{open-max}(\langle d, [min, max] \rangle) &= \langle d, [min, +\infty] \rangle \end{aligned}$$

and compute the *out* fact along the true branch of $v_1 <= v_2$ as:

$$\begin{aligned} Env_{out,T}^n(v_1) &= \text{intersect}(Env_{in}^n(v_1), \text{open-min}(Env_{in}^n(v_2))) \\ Env_{out,T}^n(v_2) &= \text{intersect}(Env_{in}^n(v_2), \text{open-max}(Env_{in}^n(v_1))) \end{aligned}$$

The $<$ comparison is treated similarly, but shifting the *min* or *max* component by one. For $!=$ comparison, in general no improvement can be made.

The *out* mapping along the false branch $Env_{out,F}^n$ is computed by evaluating the negation of the predicate. Note that if the evaluation of a predicate causes one of the predicate variables to be mapped to \top (which represents the empty range) along a given branch, then that branch is infeasible.

6.5 Widening and Narrowing

During dataflow analysis, since the interval lattice has infinite descending chains, widening [Cou⁺76] is used to ensure convergence, and narrowing is used to obtain more precise results.

The widening and narrowing operators for descriptor-offset ranges are an extension of the classic widening and narrowing operators for intervals given in [Cou⁺76]:

Widening $\langle d_1, [min_1, max_1] \rangle \nabla \langle d_2, [min_2, max_2] \rangle =$

if $d_1 \sqsubseteq_d d_2$ or $d_2 \sqsubseteq_d d_1$,

then $\langle d_1 \sqcap d_2, \left[\left(\begin{array}{cc} -\infty & \text{if } min_2 < min_1 \\ min_1 & \text{otherwise} \end{array} \right), \left(\begin{array}{cc} +\infty & \text{if } max_2 > max_1 \\ max_1 & \text{otherwise} \end{array} \right) \right] \rangle$

else \perp .

Narrowing $\langle d_1, [min_1, max_1] \rangle \Delta \langle d_2, [min_2, max_2] \rangle =$

if $d_1 \sqsubseteq_d d_2$ or $d_2 \sqsubseteq_d d_1$,

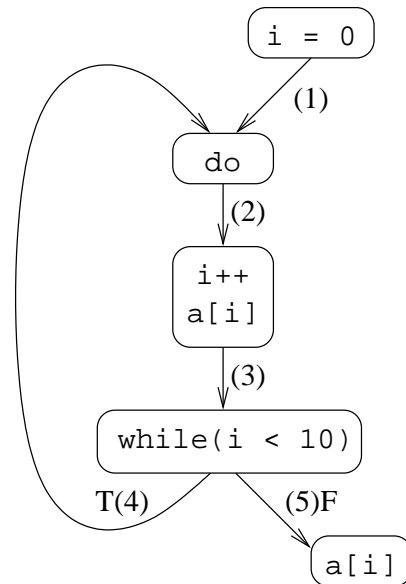
then $\langle d_1 \sqcap d_2, \left[\left(\begin{array}{cc} min_2 & \text{if } min_1 = -\infty \\ min_1 & \text{otherwise} \end{array} \right), \left(\begin{array}{cc} max_2 & \text{if } max_1 = +\infty \\ max_1 & \text{otherwise} \end{array} \right) \right] \rangle$

else \perp .

Widening is needed to ensure convergence in loops, because the interval domain has infinite descending chains, while narrowing is used to recover some precision lost due to widening.

The standard dataflow analysis worklist algorithm is performed twice:

1. “Widening” phase: dataflow facts stored along the CFG’s edges are initialized to \top . At each step in the analysis, the stored fact is combined with the newly-computed fact via the *meet* operator, except along backedges (as identified by a depth-first traversal of the CFG), in which case the stored fact is *widened* with the newly-computed fact. The analysis continues until a fixed point is reached.
2. “Narrowing” phase: starting with the stored facts computed from phase 1, the analysis is performed again, but this time, the stored fact is combined with the computed fact via the *narrowing* operator instead of the *meet*.



| | Edge (1) | Edge (2) | Edge (3) | Edge (4) | Edge (5) |
|-------------|----------|--|----------------|--|---|
| Widening : | | | | | |
| Iter. 1 | $[0, 0]$ | $\top \sqcap [0, 0]$ $= [0, 0]$ | $[1, 1]$ | $\top \nabla [1, 1]$ $= [1, 1]$ | |
| Iter. 2 | | $[0, 0] \sqcap [1, 1]$ $= [0, 1]$ | $[1, 2]$ | $[1, 1] \nabla [1, 2]$ $= [1, +\infty]$ | |
| Iter. 3 | | $[0, 1] \sqcap [1, +\infty]$ $= [0, +\infty]$ | $[1, +\infty]$ | $[1, +\infty] \nabla [1, 9]$ $= [1, +\infty]$ | $[10, +\infty]$ |
| Narrowing : | | | | | |
| Iter. 1 | | | | $[1, +\infty] \Delta [1, 9]$ $= [1, 9]$ | |
| Iter. 2 | | $[0, +\infty] \Delta [0, 9]$ $= [0, 9]$ | $[1, 10]$ | $[1, 9]$ | $[10, +\infty] \Delta [10, 10]$ $= [10, 10]$ |

Figure 6.6 Widening-Narrowing Example

Figure 6.6 gives an example demonstrating widening and narrowing on integer ranges. The table gives the ranges computed for variable `i` at each CFG edge during 3 widening iterations and 2 narrowing iterations of the analysis (the `NULL` component of the *DO* representation has been omitted for brevity). A depth-first traversal identifies (4) as a backedge at which widening will be performed at each iteration. The interesting widening operation occurs during iteration 2 at edge (4), where the upper bound of the range is widened to infinity. This guarantees or speeds up convergence: without widening it would take 10 iterations to converge, and if the `while` predicate did not constrain the range of `i`, the analysis would not converge.

But results obtained by widening can be imprecise; after iteration 3, `i` has an infinite upper bound at all CFG edges. To improve precision, another analysis pass is performed where narrowing is done at each CFG edge. The intuition behind narrowing is that an infinite bound could only have arisen as a result of over-approximation due to widening, so if starting from the infinite bounds we can derive a fact with finite bounds, the finite bound is still a safe solution. In this example, after narrowing we have a precise (finite-bounded) interval for `i` at all CFG edges.

6.6 Structure Fields

As defined, the descriptor-offset domain cannot precisely represent a pointer value that points to a field of a structure, e.g., from the assignment `p = &x.a`. This limits the ability of the analysis to infer precise ranges from predicates or pointer subtraction.

Consider the example in Figure 6.7. At line 2, `p` is assigned to point to location `s.b` plus an unknown positive offset, while at line 3, `q` is assigned to point to one past the last element of `s.b`. Because the “location” component of the *DO* domain only includes outermost objects, the field `s.b` cannot be represented, so the location must be set to `UNKNOWN`. This means the predicate at line 4 is unable to improve the range of `p` to have an offset of `[0, 9]` to determine that the dereference at line 5 is definitely in-bounds.

| | |
|--|--|
| <pre> struct S { int a; char b[10]; } s; char *p, *q; unsigned int i; 1. i = getchar(); 2. p = &s.b[i]; 3. q = &s.b[10]; 4. if(p < q){ 5. *p = ...; 6. } </pre> | <pre> i ↦ ⟨NULL, [0, +∞]⟩ p ↦ ⟨UNKNOWN : char [10], [0, +∞]⟩ q ↦ ⟨UNKNOWN : char [10], [10, 10]⟩ p ↦ ⟨UNKNOWN : char [10], [0, +∞]⟩ </pre> |
|--|--|

Figure 6.7 Structure Field Example

Note that it is unsafe to let the outermost location \mathbf{s} represent *any* field of \mathbf{s} , since it is important to ensure that if two ranges have the same target location, then they must represent offsets from the same base pointer value. Violating this property would render our handling of pointer subtraction and predicates unsound.

To handle pointers to structure fields more precisely, we define an augmented set of locations $locs_s(P)$ to include members representing structure fields, and use $locs_s(P)$ in place of $locs(P)$ in the definition of the Array Descriptor Domain on page 59. The offset of a field \mathbf{f} in structure \mathbf{s} can be portably represented as a type signature (a list of types) of the fields of \mathbf{s} up to and including \mathbf{f} . Therefore, we represent a structure field with a base location and a type list:

$$\bullet locs_s(P) = locs(P) \cup \{x.(\tau_1, \dots, \tau_n) \mid x \in locs_s(P), \tau_1, \dots, \tau_n \in T'\}$$

where T' is the set of non-void C types. This definition is recursive, to represent fields in nested structures. A union member is represented by the union location itself, because all members of a union are defined in C to have the same address as the union object itself.

For example, given the following definitions:

```

struct S {
    int i;
    int j;
    struct T {
        char c[5];
    } t;
} s1, s2[10];

union U {
    double d[8];
    struct V {
        float f;
        long l;
    } v;
} u;

```

the following table shows some locations and their representations:

| | Field | $locs_s(P)$ representative |
|----|----------------------|--|
| 1. | <code>s1.i</code> | $s1.(int)$ |
| 1. | <code>s1.j</code> | $s1.(int, int)$ |
| 2. | <code>s1.t.c</code> | $s1.(int, int, struct\{char\ [5]\}).(char\ [5])$ |
| 3. | <code>s2[i].j</code> | (UNKNOWN) |
| 4. | <code>u.d</code> | u |
| 5. | <code>u.v.l</code> | $u.(float, long)$ |

As suggested in line 3, $locs_s(P)$ does not include representatives for fields in an array of structures, but it does represent arrays within structures (as in line 2).

Using this representation, one small adjustment is needed to the definition of the *intersect* function on page 69, for handling predicates. For rule 3, the condition $x_1 \neq x_2$ is not sufficient, because if x_1 is a structure \mathbf{s} , and x_2 is a field of \mathbf{s} , then x_1 and x_2 may refer to the same location without going out of bounds. Although this can only occur in pathological cases, it is important for completeness to replace the $x_1 \neq x_2$ condition in rule 3 with $outer(x_1) \neq outer(x_2)$, where $outer(x)$ is the outermost object in $locs(P)$ containing x .

Finally, we also augment the set *Var* of variables that are mapped to ranges (defined on page 61) to include fields of structure variables. This increases the number of locations for which we compute a range, improving the likelihood of the analysis identifying in-bounds dereferences.

6.7 Evaluation

We implemented the pointer-range analysis as a context-insensitive inter-procedural dataflow analysis (operating on the supergraph of the program). After performing the

analysis, we consider each checked pointer dereference $\langle i, *q \rangle \in \text{checked-derefs}(P)$: if $q \mapsto \langle x : \tau[\sigma], [min, max] \rangle$ such that $min \geq 0$ and $max < \sigma$, then the dereference is guaranteed to be in-bounds, so $\langle i, *q \rangle$ can be excluded from $\text{checked-derefs}(P)$.

For dereferences that access a structure field (using the arrow operator), a further condition is needed to ensure that the target object contains the needed field. The dereference $sp \rightarrow j$ is guaranteed to be in-bounds if $sp \mapsto \langle x : \tau[\sigma], [min, max] \rangle$ such that $min \geq 0$ and $max < \sigma$ and τ is a structure that begins with the type signature of field j .

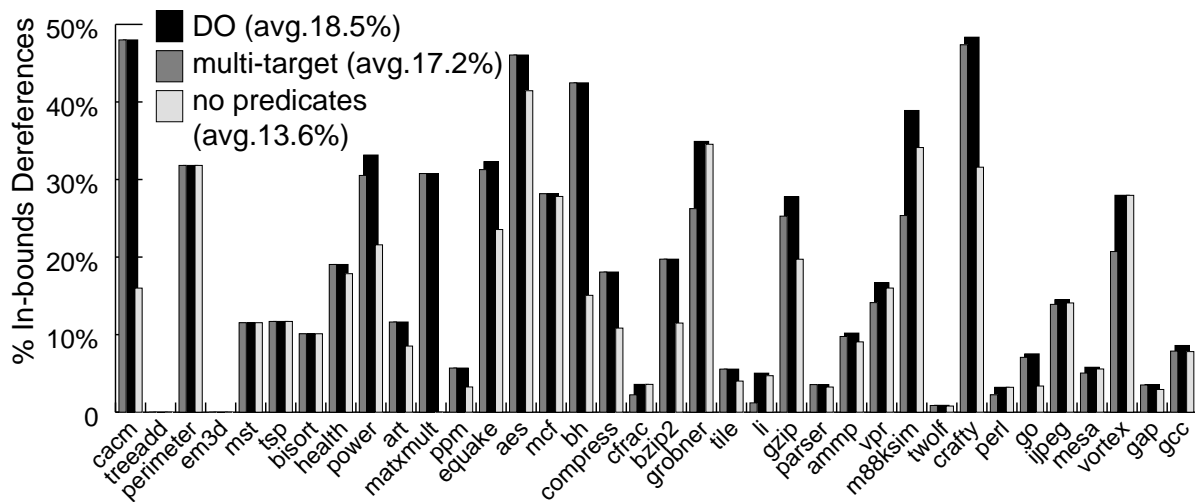


Figure 6.8 Pointer Range Analysis: In-Bounds Dereferences (Percentage)

In Figure 6.8, the black (*DO*) bars show the percentage of dereferences that were found to be in-bounds in our benchmark programs. On average, 18.5% of the dereferences were found to be in-bounds.

To justify the use of the *DO* representation rather than the simpler location-offset (*LO*) representation, we evaluated the two ways in which *DO* can give better results than *LO*:

- *multi-target*: *DO* can represent a pointer to multiple targets, as in the Figure 6.2 example.
- *transformation (6.1)*: *DO* allows the application of Transformation (6.1) or (6.1a) when handling mismatched-type operations.

We found that *multi-target* made a bigger difference; the ‘multi-target’ bars of Figure 6.8 show the percentage of dereferences found to be in-bounds when the *multi-target* ability was disabled – on average, it found 11% fewer in-bounds dereferences per benchmark. Most of these come from function calls, where different arrays of the same size are passed as an argument to a function that accesses the array. As for *transformation (6.1)*, only 35 in-bounds dereferences were not found when this feature was disabled (one in `gcc`, 21 in `m88ksim`, and 13 in `crafty`). Overall, the difference between the *DO* and *LO* is significant, and shows that the type-count descriptor is an effective mechanism for handling challenging aspects of C.

We also evaluated the importance of improving ranges at predicate nodes. The ‘no predicates’ bars in Figure 6.8 show the percentage of dereferences found to be in-bounds if ranges at predicates were not improved as described in Section 6.4. On average, 20% fewer in-bounds dereferences per benchmark were found, confirming the importance of handling predicates.

To measure the price of portability, we looked at the improvement in results if exact sizes of types are assumed, i.e., if type mismatches are handled with transformations (6.1) and (6.2) rather than (6.1a), (6.2a), and (6.2b). Only five more in-bounds dereferences were found using exact sizes (two in `gcc` and three in `gap`), suggesting that in practice, the portable transformations produce results that are almost as good as the non-portable ones.

Figure 6.9 gives the runtime overhead of MSE using the EPT classification plus pointer-range analysis, compared to using just EPT. The average slowdown with range analysis is 192% for read-write checking, and 46.5% for write-only checking (representing respectively a 6.8% and 14.0% improvement over EPT). The improvements are significant but not overwhelming, and must be weighed against the analysis complexity.

Figure 6.10 gives the analysis-time slowdown for Pointer-Range Analysis and EPT. Notice that the flow-sensitive pointer-range analysis has some scalability issues with the larger programs.

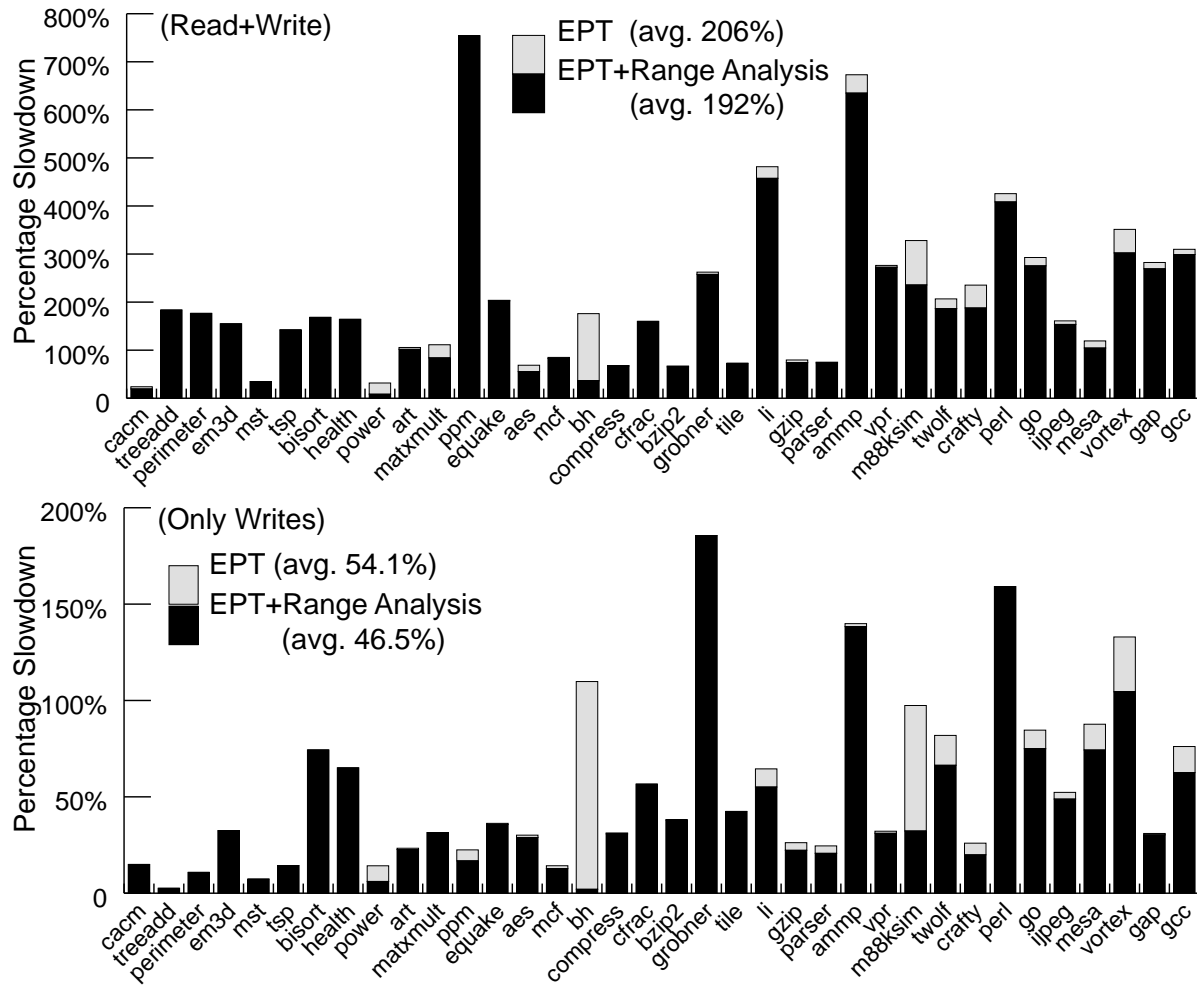


Figure 6.9 Pointer-Range Analysis: Runtime Overhead

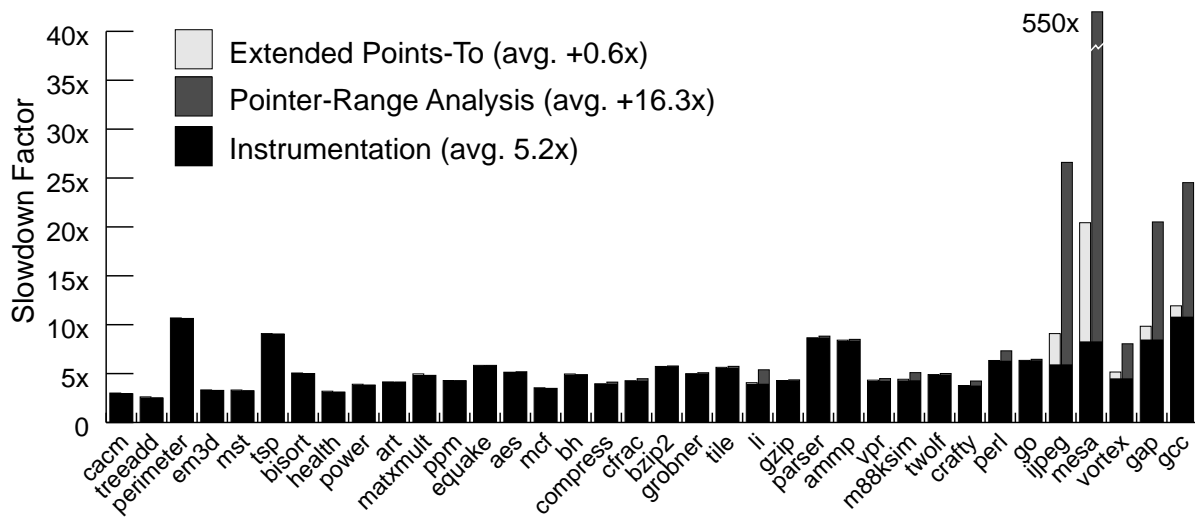


Figure 6.10 Pointer-Range Analysis: Analysis Time Slowdown

Chapter 7

Summary of Memory-Safety Enforcer

7.1 Performance and Coverage

Figure 7.1 gives the runtime overhead of MSE using EPT, EPT plus redundant checks analysis, and EPT plus both redundant checks analysis and pointer-range analysis. The average slowdown with all analyses is 170% when checking reads and writes (a 17% improvement over EPT), and 43.7% when checking writes only (a 19% improvement over EPT). Note that Redundant Checks Analysis and Pointer-Range Analysis are orthogonal, and often give complementary improvements (for example, in comparing Figures 7.1 and 6.9, `bh` shows a big improvement from pointer-range analysis but not redundant checks analysis, while `ampp` shows a big improvement from redundant checks analysis but not pointer-range analysis).

The improvements due to redundant checks analysis and pointer-range analysis are significant but not overwhelming, and must be weighed against the analysis complexity. Figure 7.2 gives the analysis times of EPT, redundant checks analysis, and pointer-range analysis, as a multiple of the compilation time of the uninstrumented program. EPT is based on a fast flow-insensitive points-to analysis that scales well to the larger programs, while the two flow-sensitive analyses are noticeably slower for the large benchmarks. The analysis of `mesa` is slow because the program has many large structures (with over a thousand fields) which our implementation currently does not handle efficiently.

Another metric that may be of interest is *tracked coverage*, or the percentage of static locations (variables, MALLOC objects, and STRLIT objects) that were classified as tracked. With the naive classification, the tracked coverage is 100%, because all user locations are

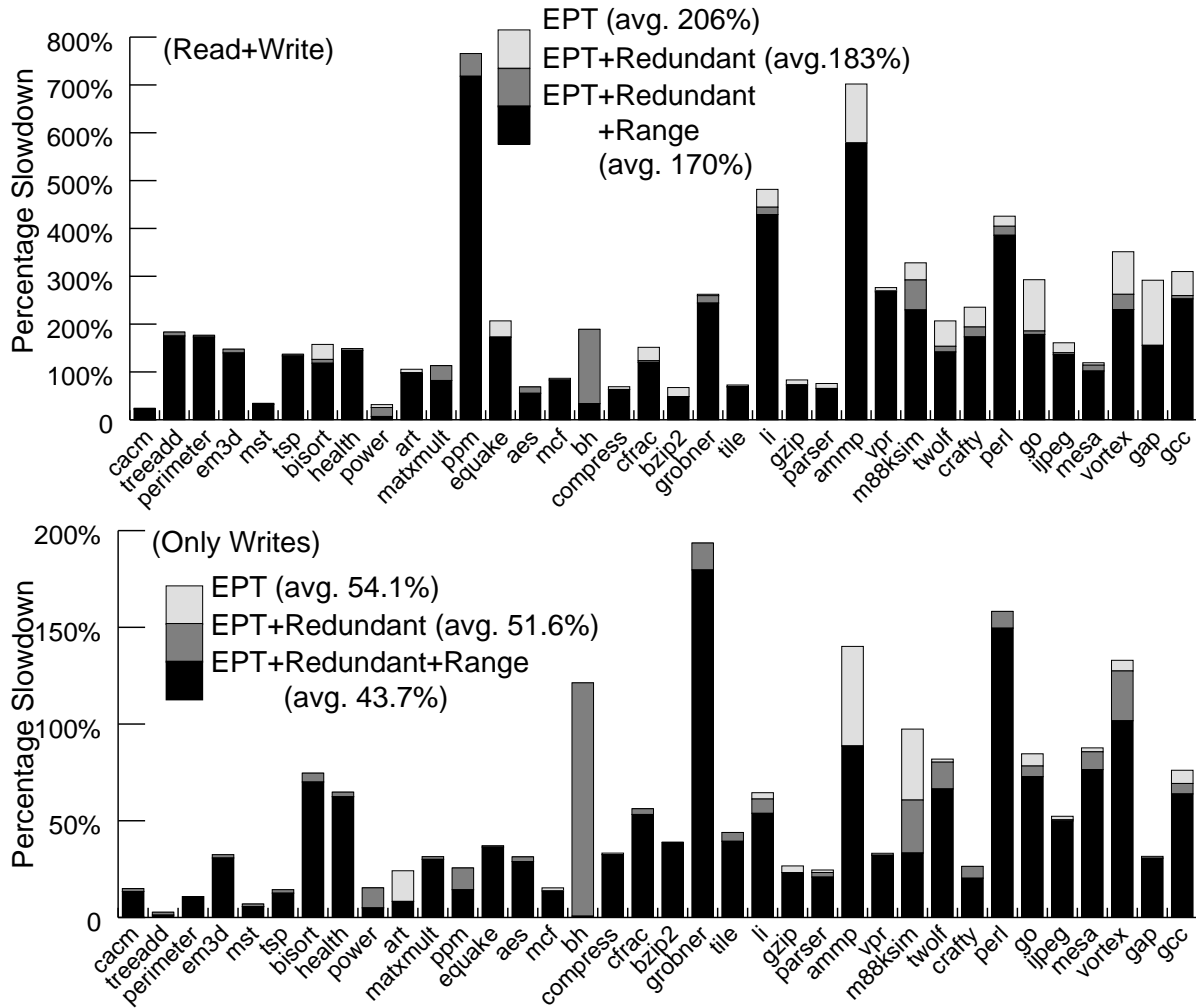


Figure 7.1 Pointer-Range Analysis: Runtime Overhead

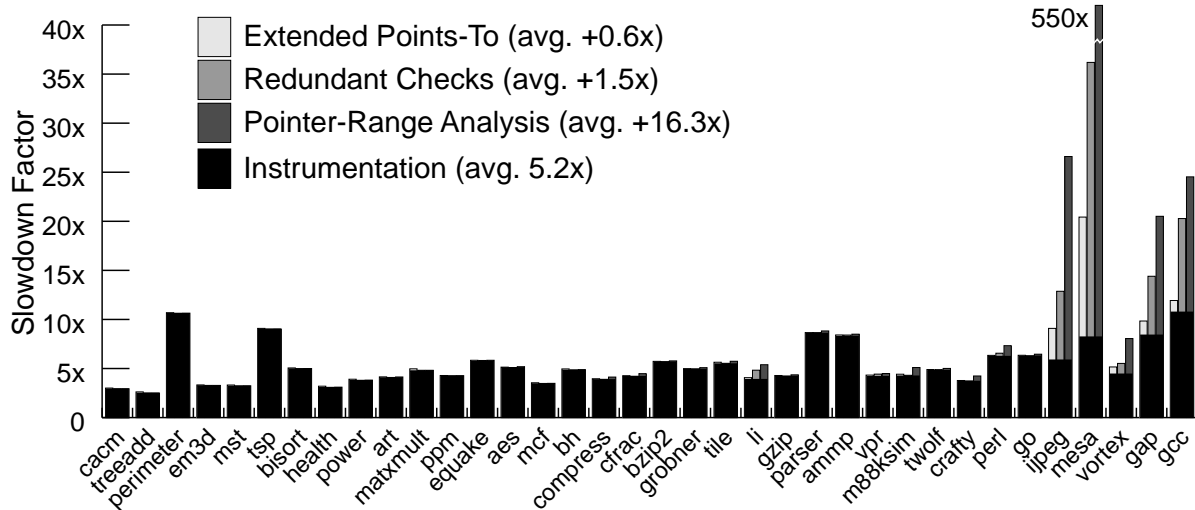


Figure 7.2 Compilation-Time Slowdown

tracked. With EPT and Range Analysis, the tracked coverage is given in Figure 7.3:¹ when checking reads and writes, the average coverage is 16.4% with EPT, and 13.6% with EPT and Range Analysis, and when checking writes only, the average is 9.7% with EPT and 7.6% with EPT and Range Analysis. The tracked coverage for `tile` is high in read-write checking mode because the program declares a large number of string literals, most of which are classified as tracked. Improvements in tracked coverage from range analysis arise if the analysis can guarantee that all accesses to an array (including via pointers) are in-bounds, and classify them as untracked. Observe that the coverage for write-only checking is lower than for read-write checking, because certain locations are only read but not written via checked dereferences. Thus, checking only writes actually increases the likelihood of detecting an invalid access, though it would fail to detect any invalid *reads*.

In the context of a security tool, a better measure of coverage is how many “sensitive” locations are tracked. Of the control-sensitive locations listed in Chapter 2, only function pointers, `longjmp` buffers, and `exec/system` call arguments are part of user memory: if any of these are classified as tracked, then an invalid write into these locations would not be

¹Recall that redundant checks analysis does not reduce the size of $tracked-locs(P)$, so there is no change in tracked coverage when using redundant checks analysis.

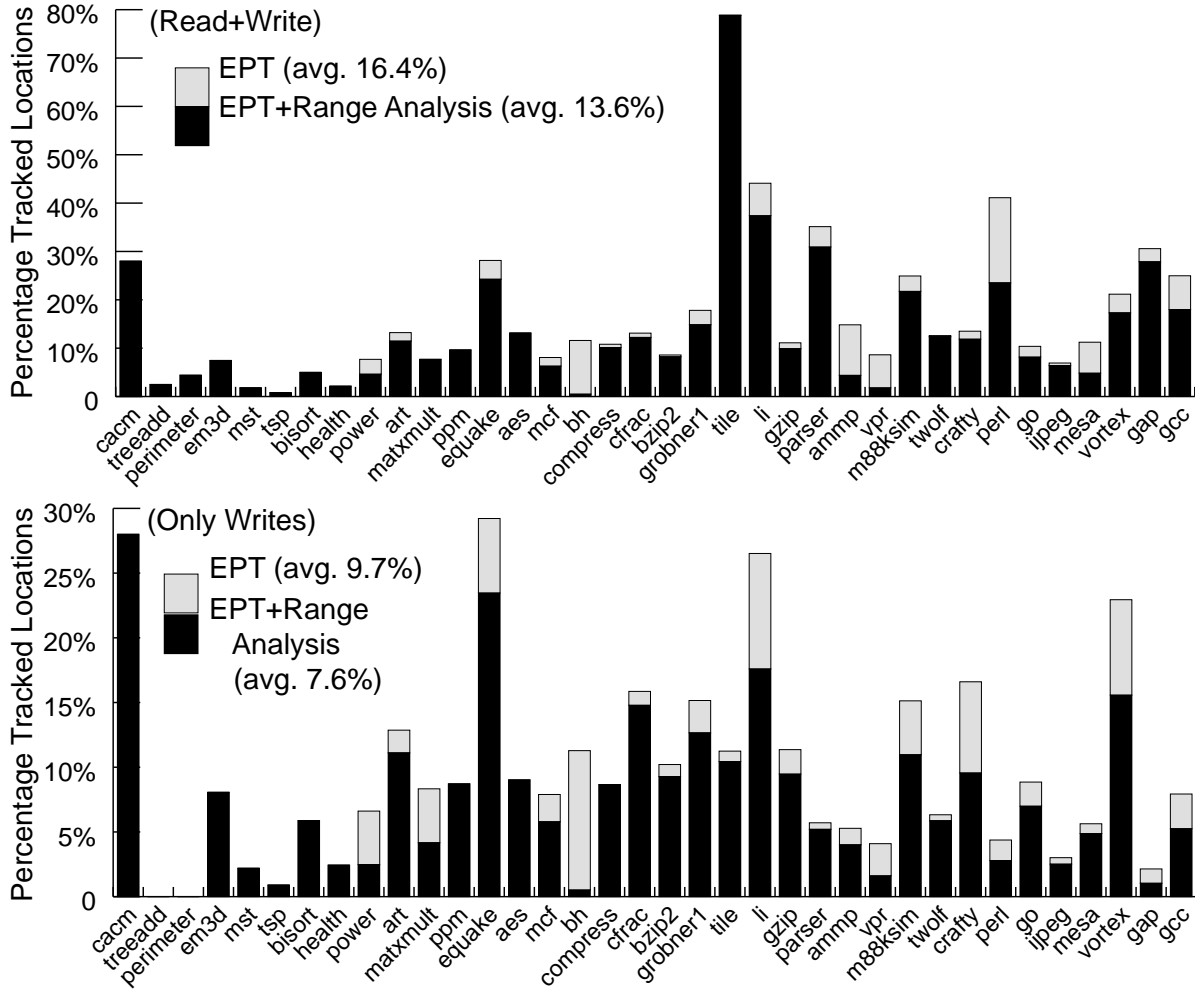


Figure 7.3 Static Tracked Coverage

detected. For all other sensitive locations, any attempt to overwrite them via a checked dereference would be detected.

Of the 151 function pointers in the benchmarks, only two are classified as tracked: these two (both in `vortex`) are tracked because they are assigned a value via a call to the library function `memcpy`, which includes a write via an unsafe dereference. Of the 11 `longjmp` buffers in the benchmarks, 9 are classified as tracked. Six of these (in `li`) are classified as tracked due to imprecise handling of structures by the points-to analysis, two (in `perl`) are arrays of `longjmp` buffers that are accessed via unsafe dereferences, and one (in `gcc`) was assigned via a call to `bcopy`, which includes a write via an unsafe dereference. Of the seven arguments to `exec` and `system` calls (in `perl`, `m88ksim`, `gcc`, and `gap`), five were classified as tracked: these were stored in arrays and manipulated by unsafe dereferences in string or array operations.

7.2 Comparison with Other Tools

Figure 7.4 gives a comparison of the execution-time slowdown of the MSE to two related tools, CCured (version 1.2.3) and Cyclone (version 0.6), for the Cyclone benchmarks. All were compiled with `-O3` optimization, and executed on the same machine on the same inputs.

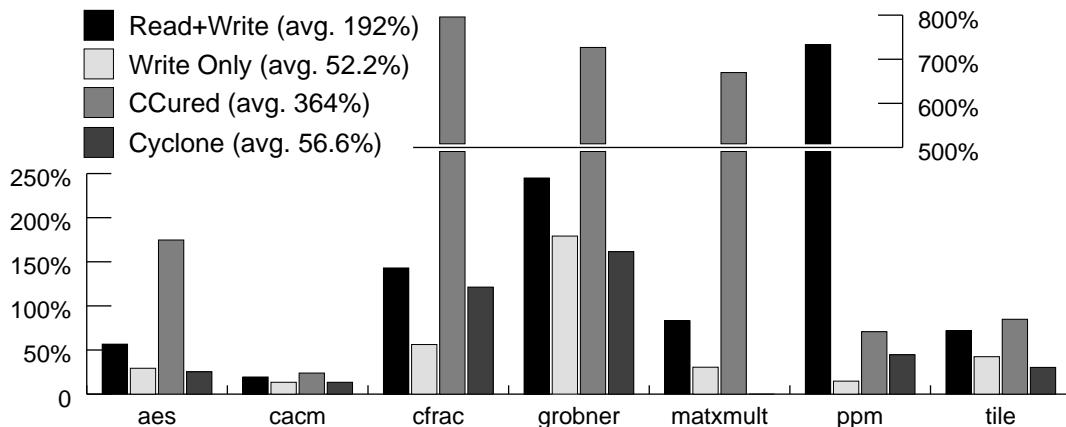


Figure 7.4 Runtime Overhead Comparison

CCured [Nec⁺02, Con⁺03] and Cyclone [Jim⁺02] are two variations of the C language that enforce memory safety (for both reads and writes) at runtime. They both make use of

fat pointers to detect spatial access errors; to prevent temporal access errors, they restrict the memory management to the use of a garbage collector (for CCured) or a region-based memory manager (for Cyclone). CCured, like the MSE, uses static analysis to classify pointers into categories of safety, instrumenting only pointers for which memory safety cannot be statically verified; additionally, user-supplied annotations can be used to improve the static analysis results. Cyclone also has different types of pointers for which different amounts of runtime checks are instrumented, but the type of each pointer is specified by the programmer. In both cases, the use of fat pointers results in a more restrictive memory-safety model; e.g., dereferencing a pointer that has been cast from an integer is not allowed.

The average slowdown for our tool on these benchmarks is 192.5% when checking reads and writes, and 52.2% when checking only writes (note that our worst-performing benchmark for checking only writes, `grobner`, is included in this test set). For CCured, the average slowdown (363.8%) is much higher than the average slowdowns reported for other test cases in their papers [Nec⁺02, Con⁺03]; the CCured team verified the poor performance for `cacm` and `matxmult` on the unmodified source files, but were able to significantly improve the performance of `cacm` with some (potentially-unsound) programmer-added annotations [Nec04]. For Cyclone (average slowdown 56.6%), the source code had been translated (manually) to Cyclone, so human intervention played a part in identifying pointers that should be made fat pointers.

Although these test cases are far from comprehensive, they demonstrate that our approach can be competitive with or faster than tools like CCured and Cyclone while maintaining the low-level control of C and without requiring programmer changes to source code.

7.3 Effectiveness of the MSE

7.3.1 Fault-Injection Study

One potential weakness of the MSE compared to a fat-pointer approach (like CCured) is that the MSE is not guaranteed to detect all memory-safety errors. If an out-of-bounds

dereference happens to access another tracked location, the dereference would not be recognized by the MSE as an invalid access. To get an idea of the extent to which this may be a problem in practice, we conducted a simple fault-injection study.

For each Cyclone benchmark, we identified all array declarations and `malloc` calls. Our plan was to create, for each array declaration or `malloc` call, a variant of the program in which the size of the array or `malloc`'d block was decreased by one; e.g., for the declaration `int a[SIZE]`, a variant would be created in which the declaration of `a` was replaced by `int a[SIZE-1]`, and for the call `malloc(e)`, a variant would be created in which the call was replaced by `malloc(e-1)`. The idea behind each of these changes was to trigger an off-by-one error. We would then instrument each variant with CCured to determine the number of cases in which a memory-access error was triggered (at runtime, on our test inputs). Since the fat-pointer approach is guaranteed to detect spatial access errors, this number could be used as a baseline against which the number of error cases detected by the MSE could be compared. However, it turned out that CCured's dynamic memory allocator increases the size of each allocated block to the next word boundary — a practice that enables efficient bounds checking — so we altered the fault-injection methodology slightly to account for this, by decreasing the size of each mutated `malloc` argument by one word rather than one byte. Figure 7.5 summarizes the test cases: column (a) lists the number of variants created and tested for each program, while column (b) shows the number of variants for which there was an invalid access (as detected by CCured²).

When instrumented with the MSE, *all* of the error cases in column (b) were detected. This was true in both read-write checking mode and write-only checking mode, and when using either the naive classification scheme, the extended points-to analysis, or the full suite of static analyses (extended points-to, redundant checks, pointer range). This experiment

²For `aes`, CCured reported two additional errors which were not memory-safety violations. In both cases, a pointer to an array of `N-1`-element arrays is passed to a function that expected a pointer to an array of `N`-element arrays; this was disallowed by CCured, even though the misaligned array declarations did not result in a subsequent invalid access.

| | (a) Variants | (b) Errors |
|----------|-----------------|---------------|
| aes | 10 | 2 |
| cacm | 4 | 3 |
| cfrac | 9 | 8 |
| grobner | 19 | 8 |
| matxmult | 2 | 2 |
| ppm | 7 | 1 |
| tile | 19 | 0 |

Figure 7.5 Fault Injection Error Cases

would tend to suggest that, in practice, the likelihood of an out-of-bounds dereference accessing another tracked location is quite low, and that, in practice, our approach is as effective as the fat-pointer approaches in detecting memory-safety errors.

7.3.2 Finding Bugs

While the MSE was designed primarily as a security tool, it can also be used for debugging. When instrumented to check both reads and writes, a number of invalid read accesses were discovered in several of the SPEC benchmarks (`compress`, `gcc`, `go`, `jpeg`, `parser`, `vortex`).³ The `gcc` bug reads via a dangling pointer dereference (pointing to a local variable in an expired scope); the rest of the errors are out-of-bounds array accesses. Though these bugs do not appear to be vulnerable to malicious attacks, this shows that our approach can also be used to detect bugs, in a spirit similar to Purify [Has⁺92], but with a lower runtime overhead.

7.3.3 Detecting Attacks

To demonstrate the efficacy of our tool, we instrumented two Linux (RedHat 6.2) programs that have known vulnerabilities and exploits. In both cases, our instrumented program detected the invalid write during a run where we attempted to perform the exploit, and halted

³The bugs in `compress`, `go`, `jpeg` were also reported in the first CCured paper [Nec⁺02].

the program before the exploit was able to gain control. Exploits were obtained from the Packet Storm website [Pac].

The first vulnerable program we tested is **traceroute**, the vulnerability and exploit for which were described in Section 2.3.

The second program, **cfingerd**, is a configurable *finger* daemon that allows each user to turn on or off the ability for others to look up information about them. A user can supply a generic message in a file `.nofinger` in their home directory that will be displayed by **cfingerd**. Version 1.4.2 of **cfingerd** has a buffer-overflow bug in the function that processes the data in the `.nofinger` file: the data is read into an 80-byte buffer with no bounds checking. An attacker can thus put a string longer than 80 characters in their `.nofinger` file to overwrite the function's return address. The exploit we tested put the shellcode within the first 80 bytes, then padded the string with a suitable number of bytes beyond 80, followed by the address of the shellcode at a position determined (by code inspection and experimentation) to coincide with the return address in the activation record. Since by default the **cfingerd** daemon is executed as root, this exploit gives the attacker root privileges on the system.

We also instrumented the 20 test cases developed by Wilander and Kamkar to evaluate dynamic buffer-overflow prevention tools [Wil⁺03]. These test cases simulate the range of possible attacks that

1. overwrite one of the following attack targets: a return address on the activation record, an “old base pointer” on the activation record, a function pointer, or a `longjmp` buffer;
2. either use a contiguous buffer overflow to write to the attack target, or use the overflow to write a pointer value to allow a re-directed write directly to the attack target;
3. overflow either a buffer on the stack or a buffer in the heap/BSS/data segment.

With the unoptimized MSE, all the attacks except those that write directly to a function pointer or a `longjmp` buffer were detected — this is as expected, since the unoptimized MSE

classifies all user locations (including function pointers and `longjmp` buffers) as *tracked*. With the EPT analysis, all 20 attacks were detected.

7.4 Conclusion

We have described the Memory-Safety Enforcer, which instruments programs to detect invalid pointer dereferences at runtime. The tagged-memory approach, which tags each byte of memory with one bit to indicate whether the byte is a valid target of a checked dereference, allows runtime checking to be efficient without restricting the flexibility of the C language or reporting false positives. While the approach is not guaranteed to prevent all invalid accesses, it will detect a large class of known security attacks, including stack smashing and the more subtle multiple-free exploit. Further, the likelihood of detecting an invalid access can be improved with better static analysis.

Three static analyses, based on Points-To Analysis, Redundant Checks Analysis, and Pointer-Range Analysis, have been described, and have been demonstrated to be effective at improving both runtime overhead and the likelihood of detecting an error. The runtime overhead of 43.7% for checking writes only is better than the performance of tools with similar goals, and should be low enough for the approach to be used in deployed software.

Chapter 8

Sensitive Location Checker (SLC)

The idea behind the Memory-Safety Enforcer (MSE) is to maximize the number of invalid accesses detected. For the purposes of security, however, not all invalid accesses are interesting. In particular, it may be sufficient to prevent the control-sensitive locations described in Chapter 2 (page 13) from being corrupted. Indeed, this is the philosophy behind StackGuard [Cow⁺00] and many related approaches [Sta00, Bar⁺00, Chi⁺01, Özd⁺02], which detect only attempts to overwrite the return address on the activation record.

In this chapter, we describe a variation of the MSE, called the Sensitive Location Checker (SLC), that focuses runtime checks to detect only invalid writes into sensitive locations. The sensitive locations include the return address, function pointers, `longjmp` buffers, and `exec/system` call arguments. The idea is to classify these sensitive locations as *tracked*, and classify as *checked* all write dereferences that cannot validly write into a sensitive location (i.e., the set of dereferences $\langle i, *p \rangle \in \text{write-derefs}(P)$ such that $\text{pt-set}(p)$ does not contain any sensitive location). The program would then be instrumented so that the following occurs at runtime:

- Initially, all memory is tagged *valid*.
- Each tracked location is tagged *invalid* when allocated, and *valid* when deallocated.
- For each checked dereference, the tag of the target location is checked: if it is *invalid*, a security violation is reported and the program is halted.

| | |
|--|--|
| <pre> 1. int a[2], i; 2. void (*fp[2])(); 3. fp[i] = &foo; 4. a[i] = 10; 5. (fp[1])(); </pre> | <pre> 1. char safe_buf[16]; 2. char vuln_buf[16]; 3. strcpy(vuln_buf, "safe_command"); 4. gets(safe_buf); 5. system(vuln_buf); </pre> |
| (a) | (b) |

Figure 8.1 Sensitive location examples.

The tag has a different meaning in this scheme than in the MSE: intuitively *invalid* means “a sensitive location”. Dereferences that may legitimately write into the tracked (sensitive) locations are not checked; this is necessary to prevent a false positive (since the sensitive locations would be tagged *invalid*) but it means that an attack that exploits an unchecked dereference would *not* be detected.

To illustrate this approach, consider the example in Figure 8.1(a): The array of function pointers `fp` is a sensitive location, so it is classified as tracked and tagged *invalid* at runtime. Other sensitive locations, like the return address on the activation record, are also tagged *invalid*. All other locations are tagged *valid*, including the untracked (non-sensitive) location `a`. The dereference `fp[i]` at line 3 may validly write into sensitive location `fp`, so it is unchecked; the dereference `a[i]` at line 4 may not validly write into any sensitive location, so it is checked. Therefore, an out-of-bounds access of `a[i]` that overwrites the sensitive location `fp` would be detected, because `fp` is tagged *invalid*. Note that this would not be detected by the MSE: the dereference `fp[i]` would be classified as checked, so `fp` would be tracked, and tagged *valid* at runtime; if `a[i]` erroneously tries to overwrites `fp`, since `fp` is tagged *valid*, no violation is reported.

The example in Figure 8.1(a) is contrived, to facilitate explanation; a more realistic example that parallels the example in Figure 8.1(a) is given in Figure 8.1(b). In this example, `vuln_buf` is a sensitive location, because it is an argument to the `system` function call, so it is classified as tracked and tagged *invalid* at runtime. The call to `strcpy` at line 3 may validly write into sensitive location `vuln_buf` so it is unchecked, while the call to `gets` at

line 4 may not validly write into any sensitive location, so it is checked. Therefore, if the call to `gets` overflows `safe_buf` and writes into `vuln_buf`, an error would be detected. Again, the MSE would not detect this attack: `vuln_buf` may be legitimately accessed via an unsafe dereference in `strcpy`, thus `vuln_buf` is tracked. This means that an erroneous write into `vuln_buf` via the call to `gets` would find `vuln_buf` tagged *valid*, and not report an error.

8.1 Effectiveness of SLC vs. MSE

The examples in Figures 8.1(a) and (b) demonstrated some attacks that would be detected by the SLC but not by the MSE. Note that there are attacks that would be detected by the MSE but not by the SLC, as in the following example.

| Program | MSE classification | SLC classification |
|--|----------------------------|---------------------------|
| 1. <code>int (*fp1)();</code> | <code>fp1</code> tracked | <code>fp1</code> tracked |
| 2. <code>int (*fp2)();</code> | <code>fp2</code> untracked | <code>fp2</code> tracked |
| 3. <code>char * p;</code> | | |
| 4. <code>p = (void *) &fp1;</code> | | |
| 5. <code>p+=4;</code> | | |
| 6. <code>*p = getchar();</code> | <code>*p</code> checked | <code>*p</code> unchecked |
| 7. <code>(*fp2)();</code> | | |

At line 4, the pointer `p` is assigned the address of `fp1`; at line 5, `p` is incremented, and at line 6, some user-controlled value is written into `*p`. If `fp1` and `fp2` were laid out such that the assignment at line 6 writes into `fp2`, then the user could supply a malicious value to change the destination of the indirect function call at line 7.

The MSE would classify `*p` as a checked dereference (because of the pointer arithmetic at line 5), and would classify `fp2` as untracked (because it is not a valid target of a checked dereference). Thus, the invalid write at line 6 would be detected. The SLC, on the other hand, would classify `*p` as unchecked because it may legitimately access the sensitive location `fp1`. This means the dereference at line 6 would not be instrumented, thus the invalid write would not be detected.

For any program, like the above example, that can be attacked in a way that would be detected by the MSE but not the SLC, one can argue — in the ideal case — that the program

is probably vulnerable to another attack that would not be detected by the MSE. Since `*p` can legitimately access the vulnerable location `fp1`, there is likely to be a way to successfully effect an attack via `*p` by overwriting `fp1` instead of `fp2`; such an attack would not violate memory safety, and thus would not be detected by the MSE. However, this argument is not as strong in practice because of imprecisions due to points-to analysis. That is, a dereference `*p` may be classified as unchecked because `p`'s points-to set contains a sensitive location, even though at runtime `*p` could never access the sensitive location.

8.2 Library Functions

The SLC must handle calls to library functions more precisely than the MSE. In the MSE, *every* write via a dereference in a library function is checked; but in the SLC, as motivated by the call to `strcpy` in the Figure 8.1(b) example, we must determine whether each call to a library function may validly write into a tracked (sensitive) location. If a call may validly write via a dereference to a sensitive location and the dereference is checked, we may report a false positive; if a library function call has a dereference that may not validly write to a sensitive location and the dereference is unchecked, we may miss detecting an invalid write to a sensitive location.

In our implementation, calls to the following library functions are handled specially by the static analysis:

```
scanf(2+), fscanf(3+), sscanf(3+),
    gets(1), fgets(1), fread(1),
    sprintf(1), snprintf(1), bcopy(2),
memset(1), memcpy(1), memccpy(1), memmove(1),
strcpy(1), strncpy(1), strcat(1), strncat(1),
```

The number in parenthesis indicates the “dereference-write” arguments, i.e., arguments that point to a location that is written by the function (e.g., the first argument to `memset` points to the destination of a dereference write; arguments 2 and above to `scanf` are pointers to the

destination of a dereference write). For each call to one of these functions, if the dereference-write arguments' points-to sets contain a sensitive location, then we leave unchanged the call to the original (uninstrumented) version of the library function. Otherwise, the call is redirected to a wrapper version of the function that performs the necessary runtime checks. In the Figure 8.1(b) example, the `strcpy` call at line 3 would remain unchanged, since the first argument may legitimately access a sensitive location (`vuln_buf`), while the `gets` call at line 4 would be redirected to a wrapper version that checks the tag of the written locations, and reports an error if it were to write to a location tagged *invalid*, like `vuln_buf`.

8.3 Implementation

The SLC was implemented by making a few adjustments to the MSE infrastructure. The first significant change is in the initial classification of checked dereferences and tracked locations. In the MSE, we first determined $checked-derefs(P)$ (via the various static analyses) before computing $tracked-locs(P)$ based on this $checked-derefs(P)$ set. In the SLC, we first specify $tracked-locs(P)$ before computing $checked-derefs(P)$ based on this $tracked-locs(P)$ set; after this step, the Extended Points-To, Redundant Checks, and Pointer-Range Analyses can be applied to trim down the $checked-derefs(P)$ set.

In the SLC, $tracked-locs(P)$ consists of the following sensitive locations in $locs(P)$:

- each function pointer that is dereferenced in an indirect call.
- for each call to `longjmp`, `_longjmp`, or `siglongjmp`, any location passed as the first argument, and any location in the points-to set of the first argument, is included in $tracked-locs(P)$.
- for each call to `system`, `popen`, or any of the `exec` functions (`execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`), any location passed as the first argument, and any location in the points-to set of the first argument, is included in $tracked-locs(P)$.

Conceptually, $tracked-locs(P)$ should also include the return address field of each activation record, the global offset table, and the `atexit` table, but since these locations are always the

same (for a given platform), and should not be manipulated directly by the user program, they need not be explicitly included in $tracked-locs(P)$.

Given the above $tracked-locs(P)$ set, the set $checked-derefs(P)$ is initialized to all dereferences that are writes ($write-derefs(P)$). Then, for each dereference $\langle i, *p \rangle$ in this set such that the points-to set of p contains a tracked location, the dereference is removed from $checked-derefs(P)$. From this initial set of $checked-derefs(P)$, the Extended Points-To, Redundant Checks, and Pointer-Range Analyses can be applied to remove checks that are guaranteed never to violate memory safety (thus, never to invalidly write into a sensitive location).

A practical change to the MSE infrastructure for the SLC was to reverse the meaning of the tag-bit value, so that 1 represents *invalid*, and 0 represents *valid*. This change allows the SLC to take advantage of the efficiency of demand-zero paging: since the entire mirror of memory needs to be initially tagged *valid*, allocating it to contain all '0's can be efficient. The tag-checking routine (which verifies that a given checked dereference does not write to a location tagged *invalid*) was adjusted to account for this change. The routines for setting and clearing the tags did not change: when a tracked location is allocated, it is tagged with the value 1 to represent *invalid*, and when it is deallocated it is tagged 0 (*valid*). This is the same behavior (at the implementation level) as in the MSE, where when a tracked location is allocated, it is tagged with the value 1 to represent *valid*, and when it is deallocated it is tagged 0 (*invalid*).

Another major change to the MSE framework is adding instrumentation to tag the sensitive non-user locations (return address, GOT, ...) as *invalid* at runtime. The most significant of these is for the return address, where code must be added to tag the return address *invalid* at the entry point of each function, and to tag it as *valid* at every exit point of the function. This aspect of the instrumentation makes the SLC approach dependent on the compiler, as there is no portable way to determine the address of the return-address field or any of the other non-user sensitive locations.

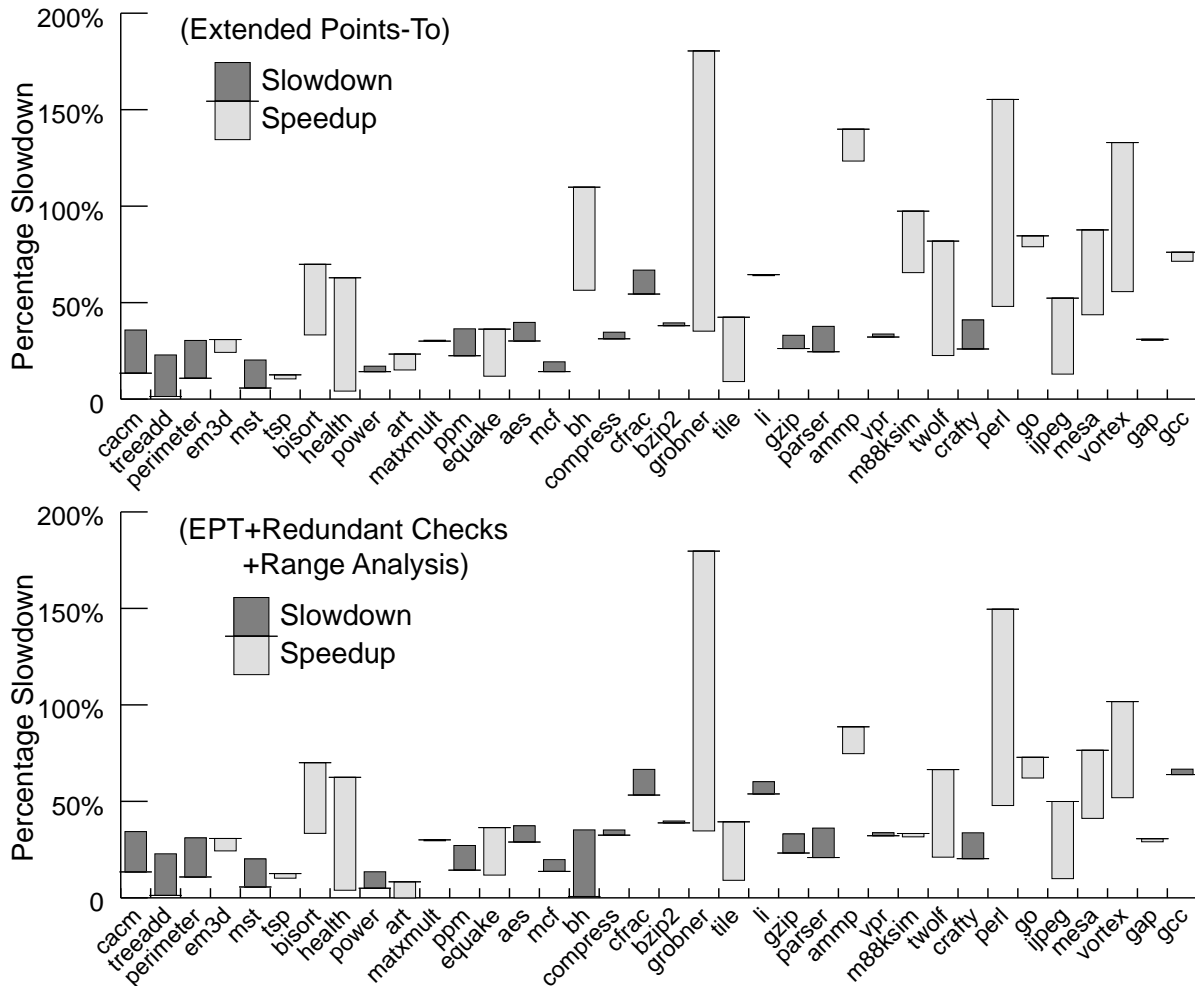


Figure 8.2 SLC vs. MSE: Runtime Overhead

8.4 Evaluation

We instrumented our benchmark programs with both SLC and MSE to compare their runtime performance. A comparison was made using just the Extended Points-To (EPT) Analysis, as well as with all three analyses (EPT, Redundant Checks, Pointer Range) enabled. Figure 8.2 compares the runtime overhead of SLC and MSE, with just EPT analysis (top graph), and with all three analyses (bottom graph). The extended horizontal lines indicate the MSE overheads, and the vertical bars indicate the amount by which SLC improved (light gray bars, pointing downwards) or worsened (dark gray bars, pointing upwards) the

runtime overhead. With EPT, the overhead of SLC was 37.7% on average, compared to MSE's 54.1%. With all three analyses, the overhead was 32.6% for SLC compared to 43.7% for MSE.

The main contributor to the difference in performance is the setting of the tags for tracked locations. In general, the SLC has many fewer tracked locations, since most programs have very few function pointers or `longjmp` and `system` calls; therefore, the SLC spends less time than the MSE in tagging tracked locations when they are allocated and deallocated. On the other hand, the SLC must instrument every function call and return to change the tag of the return-address field in the activation record, which is the main reason for the decreased performance in some of the programs compared to the MSE. (The other reason for decreased performance is the MSE optimization, described in Section 3.3, of checking large dereferences against the hash table maintained by the `malloc` wrapper routine. Since most heap locations in the SLC are not tracked, this hash table is usually empty in the SLC, so the optimization is ineffectual.)

The significant improvement in runtime performance comes at a cost: we must evaluate the likelihood that an attack will be missed by the SLC. Figure 8.3 gives the percentage of static checked dereferences in the MSE that were identified by the SLC as unchecked because they may validly refer to a sensitive location. Any attack that exploits one of these dereferences would *not* be detected by the SLC, so we want this number to be low.

In most programs, this percentage is 0; this is usually because the program does not have any sensitive user locations (i.e., they do not contain function pointers or calls to `longjmp`, `exec`, ...). However, in a few cases, the percentage of SLC-eliminated checked dereferences is discouragingly high, with 80-100% of the checks eliminated in a handful of cases. This means that almost any attack would *not* be detected for these programs. The reason for the high percentages in these cases is the imprecision introduced by points-to analysis. Most of these programs include features like indirect function calls and extensive uses of structures, which are not handled precisely by our points-to analysis. This can lead to degenerate points-to results, in which many locations can end up in the points-to set of most pointers. Thus, this

| | EPT | EPT ⁺⁺ | | EPT | EPT ⁺⁺ |
|----------|------|-------------------|-----------|------|-------------------|
| compress | 0.0 | 0.0 | aes | 0.0 | 0.0 |
| gcc | 0.2 | 0.0 | cacm | 0.0 | 0.0 |
| go | 0.0 | 0.0 | cfrac | 0.0 | 0.0 |
| jpeg | 88.6 | 93.1 | grobner | 0.0 | 0.0 |
| li | 95.0 | 100.0 | matxmult | 0.0 | 0.0 |
| m88ksim | 6.1 | 14.4 | ppm | 0.0 | 0.0 |
| perl | 95.3 | 97.9 | tile | 0.0 | 0.0 |
| vortex | 61.2 | 82.4 | | | |
| ammp | 3.6 | 3.9 | bh | 0.0 | 0.0 |
| art | 0.0 | 0.0 | bisort | 0.0 | 0.0 |
| bzip2 | 0.0 | 0.0 | em3d | 0.0 | 0.0 |
| crafty | 0.0 | 0.0 | health | 0.0 | 0.0 |
| equake | 0.0 | 0.0 | mst | 93.3 | 93.3 |
| gap | 93.3 | 94.1 | perimeter | 0.0 | 0.0 |
| gzip | 0.0 | 0.0 | power | 0.0 | 0.0 |
| mcf | 0.0 | 0.0 | treeadd | 0.0 | 0.0 |
| mesa | 74.5 | 79.2 | tsp | 0.0 | 0.0 |
| parser | 0.0 | 0.0 | | | |
| twolf | 0.0 | 0.0 | | | |
| vpr | 0.0 | 0.0 | Average | 17.0 | 18.3 |

(EPT⁺⁺ = EPT + Redundant Checks + Pointer Range Analysis.)

Figure 8.3 SLC-Eliminated Checked Dereferences (percentage, compared to MSE)

poor coverage is not a fundamental limitation of the SLC approach, and could potentially be improved with better points-to analysis.

Since the SLC and the MSE have very different notions of coverage, it is difficult to compare their relative effectiveness at detecting an attack. One point of comparison we can make is to consider the programs containing sensitive locations that were identified by MSE as tracked (as discussed in Section 7.1, these programs are `gcc`, `li`, `m88ksim`, `perl`, `vortex`, and `gap`). These are the programs for which an attack may potentially be missed by the MSE. If we look at the percentages of SLC-eliminated checks for these same programs, most are high, but two of them are low (`gcc`: 0%, `m88ksim`: 14%). For `gcc`, which had no SLC-eliminated checks, this means the SLC would be more likely detect an attack than the MSE. For `m88ksim`, security against control-transfer attacks appears to be a moot point, as the sensitive location in that program (the argument to a `system` call) can be supplied in the command-line argument; i.e., an attacker does not have to use a clever scheme that violates memory safety to “gain control” of the program.

8.5 Summary: SLC vs. MSE

We have explored two high-level approaches to protecting programs from attacks. The MSE enforces memory safety, and includes detection of memory-safety violations that might not be vulnerable to attack. It also has the potential to detect new and as yet undiscovered methods of attack, as well as less malicious attacks that may simply corrupt data to cause a denial of service, as long as such attacks violate memory safety. Because the concept of memory safety is defined by the language, the implementation can be portable, relying entirely on information available at the source level.

The SLC approach, on the other hand, targets specific locations that are known to be vulnerable to attack. In limiting the scope of the safety checks, it can achieve better runtime performance. A downside of the SLC is that the sensitive locations must be known and determined *a priori*; there may be some clever way to effect a control-transfer attack that has yet to be discovered, which may thwart the SLC protection. (As a historical note, the

format-string attack was only discovered relatively recently, in 2000. Thus, we should not be too confident in believing that we are aware of all possible means of attack.) Another downside is that many of the sensitive locations are platform dependent, thus requiring non-portable implementation. In fact, it is sometimes necessary to instrument the internals of certain library functions, to ensure that potentially unsafe dereferences within them are checked. For example, the multiple-free exploit (Section 2.3) is effected by an invalid write that occurs within the (implementation-specific) internals of the `free` function. The wrapper for `free` must account for those internal operations in order to effectively detect that attack.

Neither the SLC nor the MSE is more powerful than the other: some attacks can be detected by the SLC but not the MSE, and vice versa. However, one can decide which approach will likely be more effective by evaluating the static analysis classifications to gauge their relative coverage. For the SLC, coverage is good if the number of SLC-eliminated checked dereferences is low. For the MSE, coverage is good if the percentage of locations classified as tracked is low, and if the number of sensitive locations that are classified as tracked is low.

Chapter 9

Runtime Type Checker (RTC)

The MSE uses tagged memory to enforce memory safety, which is one of several properties that are mandated but not enforced by the C language. In this chapter, we extend the tagged memory approach to check another property that is not enforced by the C language: type safety. The approach, called the Runtime Type Checker (RTC), tags each byte of memory with four bits that encode the runtime type of the value in that byte. Whenever the value in a location is used, the runtime type encoded in the location's tag is checked against the expected type of the use. If there is a type mismatch, a type-safety *error* is reported. Assignments are treated specially: if the runtime type of a value being assigned does not match the expected type of the assignment, a type-safety *warning* is reported to indicate “suspicious type behavior” that may not be a true error. These warnings are often useful for tracking down the root cause of a subsequent error.

The RTC checks subsume the MSE checks, in that memory-safety violations that are detected by the MSE are also detected by the RTC. The RTC checks also subsume checks performed by the debugging tool Purify [Has⁺92], which uses tagged memory to check for both memory-safety violations and uninitialized memory accesses. However, the additional type information maintained by the RTC allows it to detect more subtle errors that would not be detected by the MSE or Purify — errors that manifest themselves as type-safety violations but that do not violate memory safety or access uninitialized memory.

The RTC is intended for use during program development or testing, which means a relatively high runtime overhead and a few false positives can be tolerated — though it is

important to minimize both. The core type-safety model used by the RTC, described later in this chapter, is chosen to minimize the number of false positives reported while maximizing the RTC's ability to find real errors. In the next chapter, several static analyses are described to improve the runtime overhead of the RTC.

9.1 Motivating Examples

In this section, we describe three motivating examples to illustrate the potential benefits of runtime type checking. In each case, we describe the kind of error that might be made, how the RTC would detect the error at runtime, and the interesting issues raised by the example.

9.1.1 Bad Union Access

A very simple example of a logical error that manifests itself as a bad runtime type is writing into one field of a union and then reading from another field with a different type. This is illustrated by the following code fragment:

```

1. union U {
2.     int u1;
3.     int *u2;
4. } u;
5. int *p;
6. u.u1 = 34; /* write into u.u1 */
7. p = u.u2; /* read from u.u2 — warning! */
8. *p = 0;   /* bad pointer dereference — error! */

```

In this example, an integer value is written into location `u` (at line 6), and is subsequently read as a pointer (at line 7). The value that is read from `u` is stored in variable `p`, which is then dereferenced (on line 8). The symptom of the error is the attempt to use the value 34 as an address on line 8; however, the actual point of the error can be said to be on line 7, when a value of one type is read as if it were another type (i.e., the runtime type of `u.u2` is not the same as its static type).

A tool that checks for memory-safety violations, like MSE or Purify, would report an error when line 8 is executed; however, it would not be able to point to line 7 as the source of the error, and it might not report an error at all if the value 34 happened to be a valid memory address.

The RTC would tag the single location corresponding to both `u.u1` and `u.u2` with the type of the value stored at that location. After the assignment `u.u1 = 34` on line 6, that tag would be set to *int*. At line 7, the *int*-tagged value is read and assigned via a *pointer*-typed assignment. This is a type mismatch; therefore the RTC would produce a *warning* message at line 7 indicating suspicious type behavior. The dereference at line 8 is a *use* that expects a *pointer*-typed value. Since the value in `p` is still tagged *int*, the RTC would generate an *error* message indicating a bad type use.

9.1.2 Custom Allocator

C programmers sometimes try to improve the runtime performance of memory management by writing their own custom allocator to use in place of `malloc` and `free`. For example, a programmer might allocate a large chunk of memory using a single call to `malloc` via an assignment like the following:

```
static char *myMemory = (char *)malloc(BLOCKSIZE);
```

(where `BLOCKSIZE` is some large integer value). Subsequently, when new memory is needed, a call is made to a user-defined allocation function, `myMalloc`, which returns a pointer to an appropriate part of the `myMemory` block. Similarly, calls to `free` are replaced by calls to `myFree`, which update appropriate data structures to keep track of which parts of `myMemory` are currently in use.

Consider the following code fragment:

```
1. char * cp = (int *) myMalloc(64 * sizeof(char));
2. int * ip = (int *) myMalloc(64 * sizeof(int));
   ...
3. cp[64] = 'x';
4. ip[0]++;
```

The custom allocator is used to allocate two arrays, which are both part of the same `myMalloc` block. Suppose the two arrays happen to be contiguous in memory. The dereference `cp[64]` at line 3, which is outside the bounds of the `cp` array, would erroneously write into the first byte of the `ip` array. This would corrupt the `int` value stored at `ip[0]`, and would likely cause the program to produce an incorrect output.

Because both the `cp` and `ip` arrays are part of the same `myMalloc` block of memory, the erroneous dereference `cp[64]` is within the bounds of its intended target, so it does not violate memory safety. This means that memory-safety checking approaches, including MSE, Purify, and the fat-pointer approaches, would not detect the error.

With the RTC, the assignment at line 3 causes the byte of memory at `cp[64]` to be tagged `char`. The increment operation at line 4 is a *use* of the value in `ip[0]` that expects an `int`-typed value, so the tags of the bytes of `ip[0]` are checked: in this case the tag of the first byte would contain a `char` tag, which does not match the expected `int` type; thus, an error message would be issued to indicate a bad type use.

9.1.3 Simulating Inheritance with Structures

C is not an object-oriented language, and therefore has no classes. However, programmers often try to simulate some of the features of classes using structures [Sif⁺99]. For example, the following declarations might be used to simulate the declaration of a superclass `Base` and a subclass `Sub`:

```
struct Base { int a1; int * a2; };
struct Sub  { int b1; int * b2; char b3; };
```

A function might be written to perform some operation on objects of the superclass:

```
void f ( struct Base * p ) {
    p->a1 = ...
    p->a2 = ...
}
```

and the function might be called with an actual argument either of type `struct Base *` or `struct Sub *`:

```

    struct Base base;
    struct Sub sub;
    f(&base);
    f(&sub);

```

The C language guarantees that the first field of every structure is stored at offset 0, and that if two structures have a common initial sequence — an initial sequence of one or more fields with compatible types — then corresponding fields in that initial sequence are stored at the same offsets. Thus, in this example, fields `a1` and `b1` are both guaranteed to be at offset 0, and fields `a2` and `b2` are both guaranteed to be at the same offset. Therefore, while the second call, `f(&sub)`, would cause a compile-time warning (which could be averted with an appropriate type cast), it would cause neither a compile-time error nor a runtime error, and the assignments in function `f` would correctly set the values of `sub.b1` and `sub.b2`.

However, the programmer might forget the convention that `struct Sub` is supposed to be a subclass of `struct Base`, and while making changes to the code might change the type of one of the common fields, add a new field to `struct Base` without adding the same field to `struct Sub`, or add a new field to `struct Sub` before field `b2`. For example, suppose a new `int` field, `i1` is added to `struct Sub`:

```

    struct Sub { int b1; int i1; int *b2; char b3; };

```

Now, when the second call to `f` is executed, the assignment `b->a2 = ...` would write into the `i1` field of `sub` rather than the `b2` field. The fact that the `b2` field is not correctly set by the call to `f`, or that the `i1` field is overwritten with an unintended value, will probably either lead to a runtime error later in the execution, or cause the program to produce incorrect output.

The tracking of runtime types performed by the RTC tool can detect this error. The assignment `p->a2 = ...` causes `sub.i1` to be tagged with type *pointer*. A later use of `sub.i1` in a context that expects an *int* would result in an error message due to the mismatch between the expected type (*int*) and the runtime type (*pointer*).

Note that the erroneous assignment does not violate memory safety, since `p->a2` remains within the bounds of the outermost object `sub`. Therefore, memory-safety checking approaches would not detect this error.

9.2 Type Safety

Type systems [Car97] are useful for organizing data into conceptual categories, called types, and for describing operations that are permitted on data of a given type. A good type system can improve the quality of code, make programs easier to understand and maintain, and ensure that certain classes of errors do not occur. However, a type system that is too strong can limit the expressiveness or flexibility of the language, and make it difficult to implement certain low-level tasks efficiently.

The C language and its static type system can be described as “strongly typed, modulo some loopholes”. That is, while most of the language can be statically type-checked to guarantee that type errors do not occur at runtime, this guarantee is broken by a few loopholes in the language. These loopholes are necessary to support efficient implementation and flexibility — features that are important to C programmers — and include the following:

1. Array indexing and pointer arithmetic, which may allow a dereference to access memory outside the bounds of its intended target (causing a spatial access error).
2. Storing a pointer to stack or heap objects, which allows the pointer to be dereferenced after the object is deallocated (causing a temporal access error).
3. Reading from a memory location before initializing it, so that the value read may be stale data of arbitrary type.
4. Type casting and union types, which allow any memory location to be used to store values of an arbitrary type, making it possible for an operation that expects one type to use a value of a different type.

The first two loopholes make C programs prone to memory-safety errors, while the third gives rise to uninitialized-memory-access errors — both of which can be regarded as special kinds of type errors. The fourth loophole, type casting and union types, can cause a type error to occur without violating memory safety or accessing uninitialized memory.

9.2.1 C Language and Static Type System

In this section, we review some relevant aspects of the C language and static type system, and establish some simplifying assumptions.

An *lvalue expression* is an expression that refers to a memory location. In the C language, *lvalue* expressions include variable identifiers (`x`), indirection expressions (`*e1` or `e1->mem`), array index expressions (`e1[e2]`), and field selectors `e.mem` where `e` is also an *lvalue* expression. The two contexts in C that require an *lvalue* expression are the operand of the address-of operator (`&`), and the destination of an assignment (the left-hand-side of the operators `=`, `+=`, `*=`, etc., or the sole operand of the prefix and postfix operators `++` and `--`; we shall collectively refer to these as *assignment operators*). Note that an expression having an *lvalue* does not imply that it refers to a *valid* memory location; instead, it means simply that the expression is treated (by the language semantics) as referring to a memory location.

In a C program, every expression and operator has a well-defined static type. Figure 9.1 shows a simple example program, and lists the static types of the operators occurring in the statement at line 4.¹ Each syntactic occurrence of an operator corresponds to a semantic operation whose behavior conforms to the operator's static type. For an operator with type $\tau_1 \rightarrow \tau_2$, we say the corresponding operation *uses* its (first) operand with expected type τ_1 , and *produces* a value of type τ_2 . A predicate test (in an `if` statement, `while` loop, etc.) is also considered a *use* of the predicate expression. The comma operator (`,`) is not considered a use of its first operand.

¹We use the notation $\tau\&$ to represent a τ -typed *lvalue*, or a reference to a memory location with static type τ . For example, the assignment in Figure 9.1 expects its left-hand-side argument to be a memory location with static type `float`.

| <pre> 1. int * i; 2. char * p; 3. float f; 4. f = (float) (*p + i); </pre> | |
|--|--|
| Operator (at line 4) | Static Type |
| * | $\text{char}^* \rightarrow \text{char}$ |
| + | $\text{char} \times \text{int} \rightarrow \text{int}$ |
| (float) | $\text{int} \rightarrow \text{float}$ |
| = | $\text{float} \& \times \text{float} \rightarrow \text{float}$ |

Figure 9.1 Examples of Operator Static Types.

We treat function argument passing and the return statement as assignment operations. We further assume that the assignment operation requires its source and destination operands to have matching static types (modulo type qualifiers); i.e., the = operator is assumed to have a static type of the form $\tau \& \times \tau \rightarrow \tau$. An assignment that does not satisfy this condition can be converted to one that does by adding an explicit cast to the source operand. For example, if *i* is of type `int`, the assignment `i = 'c'` can be converted to `i = (int) 'c'`.

9.2.2 Runtime Type-Safety Model

Because C does not include provisions for type-checking at runtime, we must define our own runtime type-safety model for the RTC. This type-safety model should extend C's static type system naturally, so that real errors that arise due to inappropriate uses of types are recognized as type-safety violations. On the other hand, the type-safety model should not be too restrictive. We want to avoid classifying legitimate operations as type-safety violations.

We first define the *runtime type* of a C expression as follows:

1. a. The runtime type of an *lvalue* expression *e* is the runtime type of the value last written into the location to which *e* refers.
- b. If no value has been written into that location, its runtime type is a special *uninit* type.

- c. If the location is not part of allocated memory, its runtime type is a special *unalloc* type.
2. The runtime type of a non-*lvalue* expression is the type of the value *produced* by that expression’s semantic operation (i.e., the expression’s static type).

We can define a runtime type-safety model that naturally extends C’s static type system with the following rule:

3. For each *use* of an expression e that expects a type τ , the runtime type of e should be compatible with τ .

Assignments require special attention, as they can be modeled in several ways. The RTC treats assignments in the following way:

4.
 - a. An assignment is treated as a generic memory-copy operation that does not change the runtime type of the assigned value.
 - b. An assignment is not considered a *use* of its destination operand, and the type of the value it produces is the runtime type of the assigned value rather than the static type of the assignment.
 - c. Additionally, if the runtime type of the copied value is not compatible with the expected type of the assignment, it is considered a type-safety *warning* rather than an error.

This treatment of assignments, which effectively ignores the declared type of the destination location, distinguishes the RTC from related approaches like Hobbes [Bur⁺03] and CCured [Nec⁺02]. In Hobbes, a type mismatch with the declared type of the destination location generates a warning message, while in CCured, the runtime type of the value is converted to the static type of the assignment. Legitimate programs that assign into locations declared with a different type — such as by an implementation of a custom memory allocator — would trigger a false positive in Hobbes and be restricted in behavior by CCured.

We relax our runtime type-safety model further with the following conditions that were found to reduce the number of false positives in practice:

5. The address-of operator (`&`) is not considered a *use* of its operand; i.e., taking the address of a location does not require the runtime type of the location to be well-typed.
6. All pointer types are treated as equivalent, and type qualifiers (e.g., `const`, `unsigned`) are ignored.
7. The literal value `0` (or `NULL`, or `'\0'`) is considered to be compatible with any scalar type.
8. A *use* that expects an aggregate type (a structure, union, or array) does not require the runtime type to be compatible with the expected type. In other words, our type-safety model only describes requirements for scalar types.
9. A type cast that does not change the underlying data representation, such as a cast between a pointer and an integer of the same size, does not require the type of the operand to be compatible with its expected type, and has the same runtime type as its operand. We call such a cast a *copy cast*.

A cast that changes the underlying representation, such as a cast between an `int` and a `float`, is called a *conversion cast*. A conversion cast $(\tau) e$ is considered a *use* of e which expects the static type of e , and *produces* a value of type τ .

To illustrate the treatment of copy and conversion casts, consider the example in Figure 9.2. The single location `u` is declared as a union that can be accessed as an integer (via `u.i`), pointer (via `u.p`), or float (via `u.f`). We assume that a pointer and an `int` are of the same size. At line 1, the cast from a *pointer* to an *int* is considered a copy cast because it does not change the underlying data bits; this means the value written into `u` can be legitimately used as a *pointer*, such as by the dereference in line 2. In our type-safety model, the

```

union {
    int i;
    char *p;
    float f;
} u;
1. u.i = (int) "xy";    /* copy cast */
2. putchar(*u.p);
3. u.i = (int) 6.7;    /* conversion cast */
4. u.f = 6.7;
5. u.i++;

```

Figure 9.2 Copy and Conversion Cast

value stored in `u` at line 1 would have runtime type *pointer*, which is compatible with the expected type of the *use* at line 2.

The assignments at lines 3 and 4 both appear to assign the value 6.7 to the location `u`, but they are very different semantically. At line 3, there is a conversion cast from *float* to *int* which converts the value 6.7 to the integer 6, and assigns it to the location `u`; this conversion operation is therefore treated as a *use* of 6.7 that expects a *float*-typed value and produces an *int*-typed value. At line 4, on the other hand, the bit representation of the *float* value 6.7 is copied directly into `u`; a subsequent attempt to use this value as an *int* (e.g., at line 5) would encounter a value that corresponds to the *float* encoding of the value 6.7, which would probably not be the intent of the programmer. In our type-safety model, the value stored in `u` at line 4 would have runtime type *float*, so that the *use* at line 5 which expects an *int*-typed value would be a type-safety violation.

9.3 Tracking Runtime Types

The runtime types in the RTC type system consist of the following:

- *char*, *short*, *int*, *long*, *longlong* (collectively called INTEGRAL types).
- *float*, *double* (collectively called REAL types).

- *pointer*, representing all pointer types.
- *init*, representing initialized data that is compatible with any type. The value zero has type *init*.
- *uninit*, representing uninitialized data.
- *unalloc*, representing unallocated memory.

For aggregate objects (structures and arrays), the runtime types of the component scalars are tracked independently. Enumerations are treated as *ints* (per the C specification), and *typedefs* are resolved to their underlying basic types.

At runtime, each memory location is associated with a tag that encodes its runtime type. The type tags are stored in a “mirror” of memory, with each byte mapping to a four-bit nibble in the mirror. Of these four bits, the first is a “continuation” bit that encodes the extent of the tag (0 denotes the start of a new tag, 1 denotes a “continuation” nibble), and the other three are “data bits” that encode other information. In the first nibble of a tag, the data bits encode the *type-class* of the type — one of INTEGRAL, REAL, POINTER, INIT, UNINIT, and UNALLOC. If the type is larger than one byte in size, the data bits of the second nibble encode (\log_2 of) the size of the tag. (A tag with a size of 1 can be recognized by checking that the continuation bit of the subsequent nibble is 0.) For objects larger than two bytes, the remaining data bits are currently unused (they could be used to encode information for future enhancements or optimizations).

The tags for some common scalar types are illustrated in Figure 9.3. The continuation bit needs to be set to 1 for each nibble in the entire extent of the tag, so that an access to the middle of a tag can be recognized. In fact, the encoding of the type’s size in the second nibble of the tag is redundant, as the size of the tag can be determined by counting the distance to the next nibble with a 0 continuation bit. The second nibble encoding of the size is an optimization that enables the checking of the equality of two tags (of size greater than one) to be done quickly by comparing only the first byte (two nibbles) of the tags.

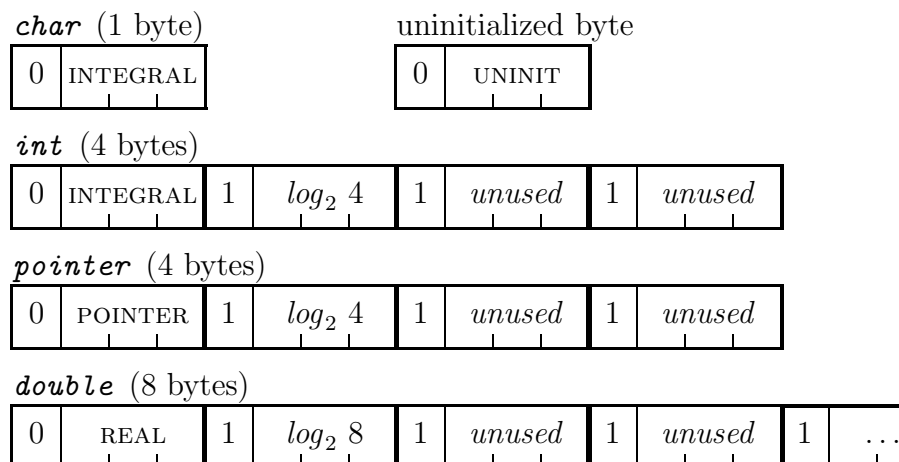


Figure 9.3 Tag Representation

Note that this tag representation captures implementation-dependent type identities; e.g., if *int* and *long* are the same size in a given implementation, their tags would be identical.

The mirror is allocated in $\frac{1}{2}$ MB pages (each mapping to 1MB of user memory). Pages are allocated on demand, and pointers to these pages are stored in a table indexed by the most significant 12 bits of the user-space address (in a 32-bit address space). Thus, looking up the tag of any given object involves a constant-time lookup in the page table followed by a constant-time lookup in the mirror page.

9.4 Instrumentation

The RTC instruments a program via a source-to-source transformation. Working at the source level gives the tool access to the static type information it needs. Programs are instrumented to perform the following actions at runtime:

- Initially, all memory is implicitly tagged with the *unalloc* type (encoded with the value 0).
- Prior to the top-level call to `main`, each global variable `v` is tagged as follows:

- If v is initialized to a non-zero value, then v is tagged with its statically-declared type.
 - Otherwise, v is tagged with the *init* type. This is consistent with the C requirement that all globals be zero-initialized.
- When a local automatic variable is declared on the stack, its tag is set to *uninit*. When exiting a function, the tags of all stack variables declared in that function are set to *unalloc*.
(Local static variables are initialized like globals, except the initialization occurs the first time the variable's scope is entered at runtime.)
- At each *use* of an *lvalue* expression e that expects a static type τ , the tag of the location e is looked up in the mirror. If the tag is not compatible with τ , i.e., if it is neither τ nor the *init* type, then a type-safety error is reported. To avoid cascading error messages, the tag of e is set to τ after an error is reported.
- At an assignment $e_1 = e_2$ that expects a scalar type τ ,
 1. The runtime type of e_2 is determined.
 2. If the runtime type of e_2 is not compatible with τ , a *warning* is reported, to signify suspicious type behavior.
 3. If e_1 is a dereference, the mirror of the location e_1 is checked: if any of it is tagged *unalloc*, a *memory-safety error* is reported.
 4. The tag of the location e_1 is set to the runtime type of e_2 .
- At an assignment $e_1 = e_2$ that expects an aggregate type τ (where τ is a structure, or a union containing a structure),
 1. If e_1 is a dereference, the mirror of the location e_1 is checked: if any of it is tagged *unalloc*, a *memory-safety error* is reported.

2. If e_2 is a dereference, the mirror of the location e_2 is checked: if any of it is tagged *unalloc*, a *memory-safety error* is reported.
3. The tags from the location e_2 are copied into the mirror of location e_1 .

The main difference with scalar-typed assignments is that the tags of e_2 's components are not checked against the components of the expected aggregate type τ .

- At a function callsite, the tags associated with the actual arguments are stored in a temporary data structure. At the entry of a function, the tags of the actuals are retrieved and assigned to the formal argument variables.

At a `return` statement, the tag of the return value (if any) is stored in the same temporary data structure; this tag is retrieved at the callsite and used as the runtime type of the function-call expression.

The interface to the RTC runtime system includes the following procedures:

- `SetTag(e, τ)`: sets the tag of location e to runtime type τ .
- `CopyTag(e_1, e_2, n)`: copies the tag(s) for n bytes of memory from the mirror of location e_2 to the mirror of location e_1 .
- `VerifyType_use(e, τ)`: verifies that the tag of location e is compatible with type τ . If the types are not compatible, a type-safety *error* is reported, and the tag of e is set to τ (to prevent cascading error messages).
- `VerifyType_assign(e, τ)`: verifies that the tag of location e is compatible with type τ . If the types are not compatible, a type-safety *warning* is reported; the tag of e is not changed.
- `VerifyDeref(e, n)`: for a dereference expression e , checks that the first n bytes of memory starting from location e are allocated (i.e., are tagged with anything other than *unalloc*); if not, a memory-safety error is reported.

| | |
|-------------------------------------|---|
| 1. <code>int i;</code> | <code>SetTag(&i, uninit_type);</code> |
| 2. <code>float f;</code> | <code>SetTag(&f, uninit_type);</code> |
| 3. <code>int *p;</code> | <code>SetTag(&p, uninit_type);</code> |
| 4. <code>i = 4;</code> | <code>SetTag(&i, int_type);</code> |
| 5. <code>p = (int *) &f;</code> | <code>SetTag(&p, pointer_type);</code> |
| 6. <code>*p = i;</code> | <code>VerifyType_assign(&i, int_type);</code> <code>VerifyType_use(&p, pointer_type);</code> <code>VerifyDeref(&*p, sizeof(int));</code> <code>CopyTag(&*p, &i, sizeof(int));</code> |
| 7. <code>i = f + 7;</code> | <code>VerifyType_use(&f, float_type);</code> <code>SetTag(&i, int_type);</code> |

Figure 9.4 RTC Instrumentation Example

These procedures are implemented using macros whenever possible to avoid the overhead of function calls.

Figure 9.4 presents a simple example program, and the RTC instrumentation that would be added at each line. For each declaration of a local variable (lines 1-3), an RTC call is added to set the variable's tag to *uninit*. At line 4, the assignment writes an *int*-typed value into *i*, so a `SetTag` call is added to set *i*'s tag to *int*. At line 5, a *pointer*-typed value is written into *p*, so a `SetTag` call is added to set *p*'s tag to *pointer*. The assignment at line 6 causes four things to happen:

1. a `VerifyType_assign` call is added to check that *i*'s runtime type is compatible with *int*; if it is not, a warning message is issued.
2. a `VerifyType_use` call is added to check that *p*'s runtime type is compatible with *pointer*; if it is not, a type-safety error message is issued.
3. a `VerifyDeref` call is added to check that **p* refers to memory that is not tagged *unalloc*; if it is, a memory-safety error is reported.
4. a `CopyTag` call is added to copy `sizeof(int)` bytes worth of tags from the mirror of *i* to the mirror of the location accessed by **p* (in this case, the mirror of *f*).

Finally, at line 7, the *use* of `f` means a `VerifyTag_use` call is added to check that `f`'s tag is compatible with `float`; in this case, `f` was tagged `int` by the assignment at line 6, so a type-safety error is reported. The `+` expression produces an `int`-typed value that is assigned into `i`, so a `SetTag` call is added to set `i`'s tag to `int`.

9.4.1 Library Functions

Certain library functions have type behavior that must be accounted for by the RTC to detect errors or to prevent the reporting of false positives. These include functions that allocate or deallocate memory (e.g., `malloc` and `free`), functions that copy memory (e.g., `strcpy` and `memcpy`), functions that read input (e.g., `fgets`), and functions that return a pointer to memory that is allocated within the library (e.g., `ctime`). Each call to one of these functions is replaced with a call to a wrapper function that performs the necessary tag manipulations to capture the type behavior of the function. The wrapper functions for `malloc` and its relatives set the tags of an allocated block to `uninit` (or, in the case of `calloc`, to `init`), while the wrapper for `free` sets the tags to `unalloc`. These wrappers also do some bookkeeping to enable `free` to know how many bytes to deallocate.

For a call to an uninstrumented function, the runtime type of the return value (if any) is assumed to be equal to its declared type. This behavior allows instrumented modules to interoperate gracefully with an uninstrumented module, which can be useful if a programmer only wants to debug one small component of a large program. Only if the function affects type behavior in a way that is externally visible (besides in the return value) would a wrapper function need to be written to prevent false positives from being reported.

9.5 Experience with Finding Bugs

To test the effectiveness of the RTC as a debugging tool, we used Fuzz [Mil⁺95] to find Solaris utilities that crash on some random input, and instrumented five such programs for testing (`nroff`, `plot`, `ul`, `units`, `col`). In each case, the RTC reported useful error messages before the program crashed. Furthermore, the tool detected a number of bugs in the Spec 95

(`go`, `jpeg`, `vortex`) and Spec 2000 (`mesa`, `parser`) benchmarks. To help determine the root cause of each error, the RTC can be used in conjunction with an interactive debugger like GDB [Sta⁺02]. When a warning or error message is issued, a signal (`SIGUSR1`) is sent, which can be intercepted by GDB; the user is then able to use the facilities of GDB to examine values in memory, including the RTC mirror, to track down the cause of an error.

All of the errors we detected were either memory-safety errors or uses of uninitialized variables, i.e., errors that could be detected by Purify. However, in three of the cases (`nroff`, `vortex`, and `parser`) the RTC detected some invalid accesses which happened to land in allocated memory but accessed data of a different type — these invalid accesses would not be detected by Purify. For example, in `nroff`, an array of pointers is erroneously accessed with a negative index. The retrieved value, when dereferenced, causes a segmentation fault. It turns out the erroneous array access reads a word from an array of characters declared elsewhere in the program. Thus, the erroneous array access causes the RTC to report a type-safety warning, because the tags of the read value (a series of *chars*) do not match the expected type (*pointer*). Purify does not detect the invalid array access, and only reports an error when the erroneously-read value is dereferenced. Thus, the RTC warning identifies the point of the program at which the cause of the error (the erroneous array access) occurs; furthermore, if the value read from the erroneous array access happened to land in allocated memory, the subsequent invalid dereference might not be detected by Purify.

We can easily create examples (such as the ones given in Section 9.1) for which the RTC would report errors that are *not* detected by Purify; however, we have not yet found examples of those kinds of bugs in real programs. We believe such bugs are more likely to arise during the software-development cycle, and we have not had the opportunity to experiment with the RTC in such a setting. The programs we have used to date for testing are in most cases robust code that has been in use for some time.

9.5.1 False Positives

In our experiments, we have found that the RTC reports very few false positives. The cases we encountered of legitimate operations that trigger RTC warnings or errors include the following:

- **Byte-wise copying:** A function that copies arbitrary data from one location to another is usually implemented such that the copy is performed one byte at a time (using a `char` pointer). Thus, each assignment expects a `char`, which causes the RTC to report a warning if the underlying data is of a different type.

A shortcoming of our current tag encoding scheme is that it does not enable us to encode pieces of a large tag and faithfully reconstruct the full tag from the pieces. Thus, if the program copies an `int` value (a four-byte INTEGRAL tag) one byte at a time, our tag-manipulation routine would produce four successive `char` tags (four one-byte INTEGRAL tags).

- **Data overlays:** In programs that are interested in the machine representation of data, such as in systems or networking, the same series of bytes may be operated on using different data overlays. A common example in networking is the IP address, which may be treated as an array of four bytes, or as a single 32-bit word. If the data is written as one type and used in a context that expects a different type, an RTC error would be reported.
- **Hash computation:** When computing a hash value for a pointer, the `pointer`-typed value (which has been *copy-cast* to an `int`) is used by an arithmetic operation that expects an `int`-typed value, which triggers the RTC to report a type-safety error.

With the diagnostic messages reported by the RTC (which include the file name and line number, the offending C expression, and the RTC tags), it is usually quite straightforward to recognize an error message as a false positive. The RTC includes a facility to suppress output of duplicate messages (messages of the same kind originating from the same file and line number), or to filter out certain error messages completely.

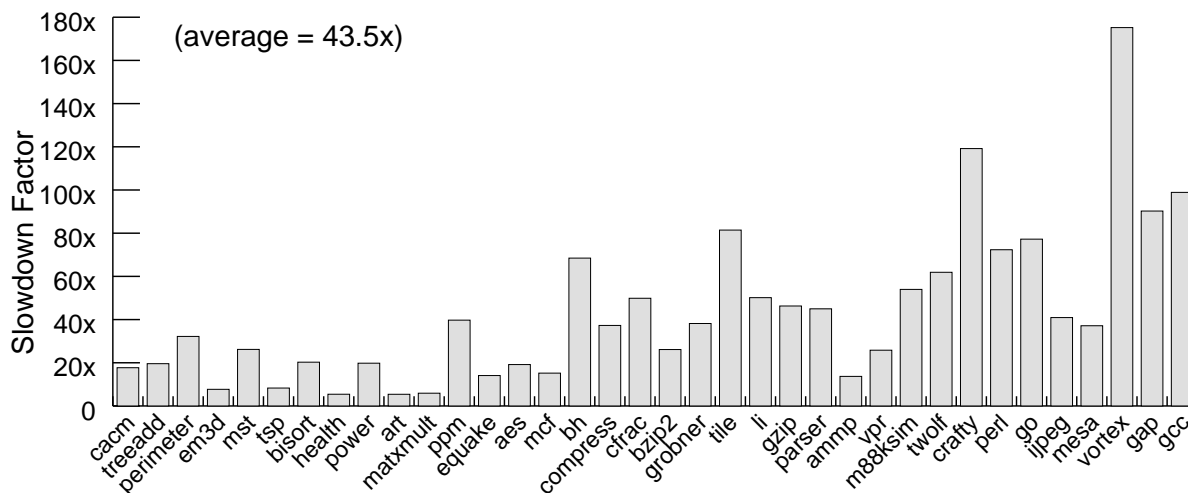


Figure 9.5 RTC Runtime Overhead

9.6 Performance Evaluation

Figure 9.5 gives the runtime overhead of the RTC, as the ratio of the running time of the RTC-instrumented executable over the running time of the original (uninstrumented) executable. The average is $44\times$ slowdown, which is quite high, but acceptable in a debugging setting.

Figure 9.6 gives the slowdown in compilation time, comparing the time it takes to instrument and compile a program with RTC to the time it takes to compile the original uninstrumented program. Recall that the RTC is a source-to-source transformation, and the instrumented procedures are implemented as macros whenever possible; thus, the instrumented source file is considerably larger and more complicated (with nested expressions, etc.). Nonetheless, the compilation slowdown (average $17\times$) is still very reasonable; more importantly, the approach scales to large programs.

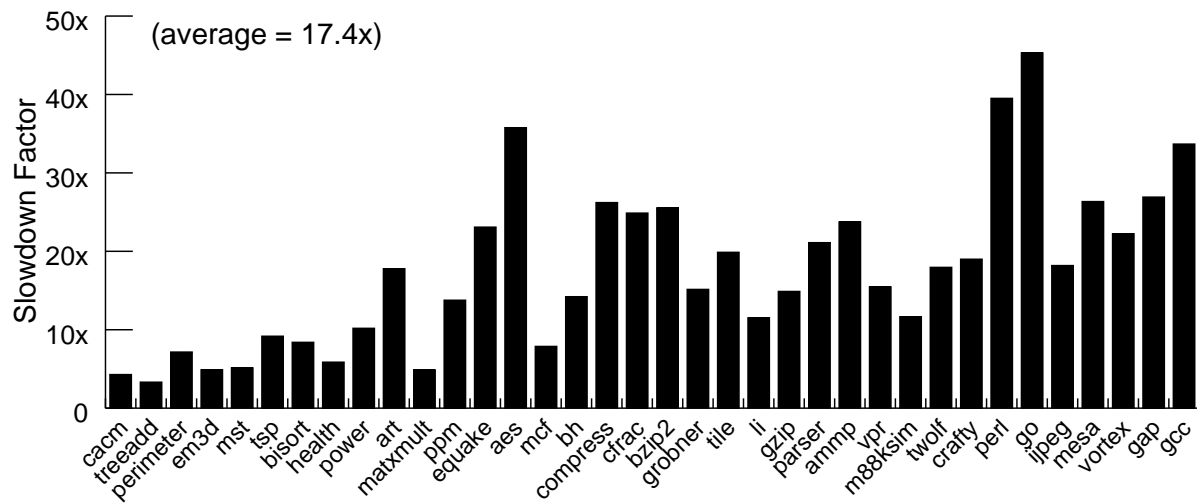


Figure 9.6 Compilation-Time Slowdown

Chapter 10

Improving the Runtime Type Checker

The runtime overhead of the unoptimized RTC is quite high, because it instruments *every* use of a memory location in the program and tags *every* user-defined memory location with a runtime type. Although this may be acceptable in some debugging settings, there is room for improvement. If used in program testing, improvements to the runtime overhead can translate to the ability to run more test cases, thus increasing the test coverage.

We first describe a flow-insensitive analysis, called *Type-Flow Analysis*, whose goal is to identify locations and expressions that are guaranteed never to violate memory or type safety, and thus need not be instrumented with runtime checks. The analysis first makes some unsafe assumptions with respect to temporal access errors and uses of uninitialized values. We then address the first problem by treating any pointer to a stack or freed-heap location as unsafe, and address the second problem with a flow-sensitive May-Be-Uninitialized Analysis. Finally, we describe Redundant Checks Analysis, which eliminates checks of expressions that are guaranteed to have already been checked.

10.1 Type-Flow Analysis

Type-Flow Analysis is a flow-insensitive analysis that traces the flow of types through memory locations in the program, to compute the types of values that may be encountered at runtime for a given *lvalue* expression or memory location. The analysis unsafely assumes that temporal access errors and uninitialized-memory-access errors never occur, because

accounting for these possibilities would require flow-sensitivity to obtain any meaningful improvements. We address these concerns later in Sections 10.2 and 10.3.

The type-flow analysis computes two classifications. First, it classifies each *lvalue* expression into one of the following categories:

- **safe**: an expression that never causes a spatial access error, and whose runtime type is guaranteed always to be compatible with its static type. All runtime instrumentation is eliminated for safe expressions.
- **type-unsafe**: an expression that never causes a spatial access error, but whose runtime type may be incompatible with its static type. Each *use* of a type-unsafe expression is instrumented to check its runtime type.
- **mem-unsafe**: an expression that may cause a spatial access error, and which must be instrumented to check for both type-safety and memory-safety.

Next, the analysis classifies each location into one of the following categories:

- **untracked**: A location that may only be accessed by safe expressions. An untracked location need not be tagged with its runtime type.
- **tracked**: A location that always contains a value of the same runtime type τ , but may be accessed by a type-unsafe or mem-unsafe expression. A tracked location is tagged with its statically-determined type τ (usually its static type) when it is declared. The tag of a tracked location remains unchanged throughout program execution.
- **unsafe**: A location that may cause a type-safety violation at runtime. An unsafe location is tagged with the type *uninit* when it is declared. The tag of a tracked location will be updated to reflect its runtime type, which may change during program execution.

In this section, we use the term *location* to refer to an abstract “outermost” object, which includes variables, and MALLOC and STRLIT objects; i.e., elements of the *locs* set defined in

Section 3.4, page 30. An occurrence of a call to `malloc` at program point i is treated as equivalent to the expression $\&\text{MALLOC}_i$, while an occurrence of a string literal at program point i is treated as equivalent to the expression $\&\text{STRLIT}_i$. In the type-flow analysis, each aggregate object (structure, union, or array) is conservatively treated as a single object representing all fields or elements of the object. Therefore, field selectors can be ignored (i.e., `x.mem` is converted to `x`). Also, we assume that all dereferences are of the form `*p` where `p` is a pointer variable. With structure fields ignored, the other forms of dereferences can be normalized to the form `*p` with the suitable introduction of temporary variables. Given these assumptions, *lvalue* expressions include only variables and dereferences of the form `*p`.

Figure 10.1 (a) presents an example code fragment to illustrate the intuition behind the approach. Since the approach is flow-insensitive, the order of the statements is ignored in the analysis.

Figures 10.1 (b) and (c) give the type-safety categories we wish to identify for the expressions and locations in the example program. The expressions `i1`, `i2`, and `i3` are safe because they are only assigned *int*-typed values. Likewise, the expressions `p1`, `p2`, and `p3` are *safe* because they are only assigned pointer-typed values (recall that the RTC tool does not differentiate between pointer types, so the fact that `p2` is assigned both the address of an *int* variable and the address of a *float* variable is not important). The expression `*p1` is safe because it only accesses variable `i1`, which always contains a value of type *int*, which matches the static type of `*p1`.

The expressions `f2` and `*p2` are *type-unsafe*. Variable `f2` is *type-unsafe* because it may contain either a *float*-typed value (assigned at line 4) or an *int*-typed value (assigned at line 9 via `*p2`), while the expression `*p2` is *type-unsafe* because it may refer to a *float*-typed value (in `f2`) which is incompatible with its statically expected type (*int*). The expression `*p3` is *mem-unsafe* because it may refer to an out-of-bounds address (as a result of the pointer arithmetic at line 10).

| | |
|--|--|
| (a) Code | (b) <i>lvalue</i> expressions |
| <pre> int i1; int * p1; int i2; float f2; int * p2; int i3; int * p3; </pre> | <pre> safe: i1, i2, i3, p1, p2, p3, *p1 type-unsafe: f2, *p2 mem-unsafe: *p3 </pre> |
| | (c) Locations |
| <pre> 1. p1 = &i1; 2. *p1 = 23; 3. i2 = *p1; 4. f2 = 6.7; 5. if(*p1 == 0) 6. p2 = &i2; 7. else 8. p2 = (int *) &f2; 9. *p2 = 4; 10. p3 = &i3 + 1; 11. *p3 = i1; </pre> | <pre> untracked: i1, p1, p2, p3 tracked: i2, i3 unsafe: f2 </pre> |
| | (d) Assignment Edges |
| | <pre> p1 ←= VALUE_{valid-ptr} *p1 ←= VALUE_{int} i2 ←= *p1 f2 ←= VALUE_{float} p2 ←= VALUE_{valid-ptr} p2 ←= VALUE_{valid-ptr} *p2 ←= VALUE_{int} p3 ←= VALUE_{pointer} *p3 ←= i1 </pre> |

Figure 10.1 Type-flow example.

Of the safe locations, `i2` and `i3` are *tracked* because, although they will always contain an *int*-typed value, `i2` may be accessed by the *type-unsafe* expression `*p2` and `i3` may be legitimately accessed by the *mem-unsafe* expression `*p3`. Since every use of `*p2` and `*p3` will be instrumented to check the runtime type of the location they refer to, each such location — including `i2` and `i3` — must be tagged with its runtime type so as not to trigger a false positive when checked via `*p2` or `*p3`. For the *untracked* locations, on the other hand, their runtime types are never checked, so their mirror need not be tagged with their runtime types (i.e., they are tagged *unalloc*).

10.1.1 Analysis Outline

The type-flow analysis involves the following steps:

1. Perform points-to analysis, to compute a points-to set $pt\text{-}set(x)$ that maps each location x to a set of locations to which x may point. (See Chapter 4 for an overview of points-to analysis).
2. Build an *assignment graph* in which nodes represent locations, *lvalue* expressions, and values with a known type, and edges represent the flow of runtime types due to assignments.
3. Compute two attributes, *required-type* and *possible-type*, for each node in the graph.
4. Categorize each *lvalue* node in the graph as either *safe*, *type-unsafe*, or *mem-unsafe*, and classify locations as either *untracked*, *tracked*, or *unsafe*.

10.1.2 Building the Assignment Graph

The assignment graph records the possible flow of runtime types among the expressions and locations in the program. The graph contains three kinds of nodes:

- **Locations:** variables, MALLOC and STRLIT objects.
- **Dereferences:** `*p` where `p` is a pointer.

- **Values:** nodes of the form VALUE_τ representing a value with statically-known type $\tau \in T^\#$.

The set of types $T^\#$ used in the type-flow analysis differs slightly from the set of types used at runtime. $T^\#$ includes the following:

- *char*, *short*, *int*, *long*, *longlong* (INTEGRAL types).
- *float*, *double* (REAL types).
- *init*, representing the value 0.
- *valid-ptr*, representing a pointer value that will not cause a spatial access error. For example, the expression $\&x$ has type *valid-ptr*.
- *pointer*, representing a pointer value that may evaluate to an invalid address. For example, the expression $\&x + k$ has type *pointer*.

The differences between $T^\#$ and the RTC runtime types are the exclusion of the *uninit* and *unalloc* types, and the refinement of *pointer* into two types, *valid-ptr* and *pointer*.

Edges in the assignment graph are directed, and represent assignments in the program. For each assignment $e_1 = e_2$ in the program,

- If e_2 is an *lvalue* expression, or a copy-cast of an *lvalue* expression, an assignment edge is added to the assignment graph leading from the node representing e_2 to the node representing e_1 .
- If e_2 is a non-*lvalue* expression with static type τ , an assignment edge is added to the assignment graph connecting VALUE_τ to the node representing e_1 .

Figure 10.1 (d) shows the assignment edges in the assignment graph for the corresponding assignment statements in column (a). Note that, at line 8, the expression $\&f2$ is treated as having static type *valid-ptr*, and the copy cast (at line 8) is effectively ignored.

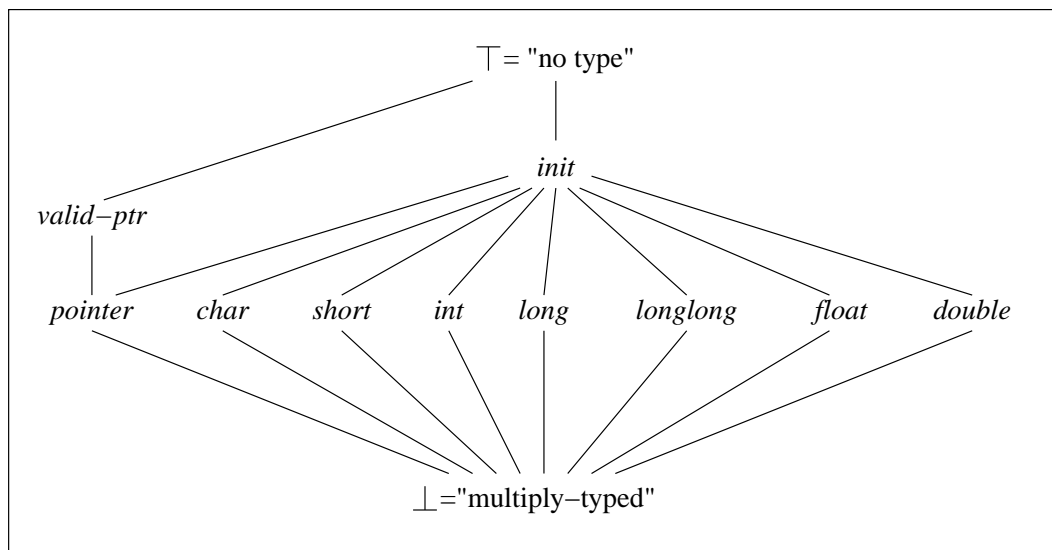


Figure 10.2 The lattice containing $T^\#$.

10.1.3 Computing Possible and Expected Types

After building the assignment graph, the analysis computes two attributes for each node n in the graph:

- $possible\text{-}type(n)$, represents the set of types of the values to which n may evaluate at runtime.
- $required\text{-}type(n)$, represents the type(s) that would satisfy type-safety for *all* uses of n in the program.

The values of these attributes are an extension of $T^\#$ to form the lattice shown in Figure 10.2. The bottom element (\perp) represents more than one incompatible type. Note that, for example, the types *int* and *long* are treated as distinct types, even though they may be identical at runtime for a given implementation. Keeping these types distinct is a safe, portable approximation.

Figure 10.3 gives the constraints for computing $required\text{-}type$ and $possible\text{-}type$ for each node in the assignment graph. For each node, the $required\text{-}type$ attribute is the least value

| | Condition | Constraint |
|-----|---|--|
| R0. | | $required\text{-}type(n) \sqsupseteq \perp$ |
| R1. | e is an <i>lvalue</i> expression, and e is used or assigned with expected type τ | $required\text{-}type(e) \sqsupseteq \tau$ |
| R2. | p is dereferenced | $required\text{-}type(p) \sqsupseteq valid\text{-}ptr$ |
| P0. | | $possible\text{-}type(n) \sqsubseteq \top$ |
| P1. | | $possible\text{-}type(VALUE_\tau) = \tau$ |
| P2. | $x \in pt\text{-}set(p)$ | $possible\text{-}type(*p) \sqsubseteq possible\text{-}type(x)$ |
| P3. | $*p \stackrel{=}{\leftarrow} n,$ $x \in pt\text{-}set(p)$ | $possible\text{-}type(x) \sqsubseteq possible\text{-}type(n)$ |
| P4. | $v \stackrel{=}{\leftarrow} n,$ v is a variable | $possible\text{-}type(v) \sqsubseteq possible\text{-}type(n)$ |

Figure 10.3 Rules for computing *required-type* and *possible-type*.

(in the lattice) satisfying the constraints introduced by rules R0-R2, while the *possible-type* attribute is the greatest value satisfying the constraints introduced by rules P0-P4. Rules R0 and P0 set the ground constraints for each node.

Rule R1 constrains the *required-type* of an *lvalue* e to be no lower than the expected types of each use of e or assignment into e . Rule R2 treats each dereference $*p$ as a *use* of p that expects the *valid-ptr* type, and constrains the *required-type* of p to be no lower than *valid-ptr*. In most cases, the *required-type* of an *lvalue* e should be equal to the static type of e , since the uses of e will expect the static type of e . Exceptions to this condition arise due to our imprecise handling of aggregate objects, where the *lvalues* $x.m$ and $x.n$ are both mapped to the *lvalue* x ; thus, if $x.m$ and $x.n$ have incompatible static types, and both are used or assigned, then the *required-type* of x will be *init* or \top .

For *possible-type*, Rule P1 sets the *possible-type* of each $VALUE_\tau$ node to its type τ . Rule P2 constrains the *possible-type* of a dereference node $*p$ to be no higher in the lattice than the *possible-type* of each location in p 's points-to set. Rules P3 and P4 handle assignment edges: if the left-hand side is a dereference $*p$ (rule P3), then the *possible-type* of each node

in the points-to set of p can be no higher than the *possible-type* of the right-hand side; if the left-hand side is a variable (rule P4), then its *possible-type* can be no higher than the *possible-type* of the right-hand side.

The *possible-type* constraints are solved by building a directed graph with the same nodes as the assignment graph, and edges representing \sqsubseteq constraints induced by rules P0-P4, with cycles collapsed for efficiency. Initially, each VALUE node is given a *possible-type* matching its static type (per rule P1), and all other nodes are given *possible-type* = \top (per rule P0). The graph is then traversed to propagate *possible-type* values along the \sqsubseteq edges: for each node n , *possible-type*(n) is assigned the meet of the old value of *possible-type*(n) with the *possible-type* of all n' for which there is a constraint edge representing $n \sqsubseteq n'$.

Figure 10.4 shows the assignment edges, inferred constraints, and final *possible-type* attributes for the example of Figure 10.1. The *required-types* are also given: the three pointers that are dereferenced have a *required-type* of *valid-ptr*; for the other expressions, the required type is the same as the static type.

10.1.4 Categorizing Expressions and Locations

Once the *required-type* and *possible-type* attributes have been computed for each node in the graph, we can assign the *lvalue* expressions and locations into the appropriate categories. First, the *lvalue* expressions are classified into *safe*, *type-unsafe*, or *mem-unsafe*, then the locations are classified (into *untracked*, *tracked*, and *unsafe*).

For each *lvalue* expression e :

- L1. If e is of the form $*p$, and *required-type*(p) = *valid-ptr*, and *possible-type*(p) \neq *valid-ptr*, then e is *mem-unsafe*;
- L2. Else, if *possible-type*(e) $\not\sqsupseteq$ *required-type*(e), then e is *type-unsafe*;
- L3. Otherwise, e is *safe*.

| Assignments | Assignment Edges | Inferred Constraints |
|------------------------|--|--|
| p1 = &i1; *p1 = 23; | p1 $\xleftarrow{=}$ VALUE _{valid-ptr} | possible-type(p1) \sqsubseteq valid-ptr |
| | *p1 $\xleftarrow{=}$ VALUE _{int} (pt-set(p1) = {i1}) | possible-type(i1) \sqsubseteq int |
| i2 = *p1; | i2 $\xleftarrow{=}$ *p1 (pt-set(p1) = {i1}) | possible-type(i2) \sqsubseteq possible-type(*p1) possible-type(*p1) \sqsubseteq possible-type(i1) |
| f2 = 6.7; | f2 $\xleftarrow{=}$ VALUE _{float} | possible-type(f2) \sqsubseteq float |
| p2 = &i2; | p2 $\xleftarrow{=}$ VALUE _{valid-ptr} | possible-type(p2) \sqsubseteq valid-ptr |
| p2 = (int *) &f2; | p2 $\xleftarrow{=}$ VALUE _{valid-ptr} | possible-type(p2) \sqsubseteq valid-ptr |
| *p2 = 4; | *p2 $\xleftarrow{=}$ VALUE _{int} (pt-set(p2) = {i2, f2}) | possible-type(i2) \sqsubseteq int possible-type(f2) \sqsubseteq int possible-type(*p2) \sqsubseteq possible-type(i2) possible-type(*p2) \sqsubseteq possible-type(f2) |
| p3 = &i3 + 1; | p3 $\xleftarrow{=}$ VALUE _{pointer} | possible-type(p3) \sqsubseteq pointer |
| *p3 = i1; | *p3 $\xleftarrow{=}$ i1 (pt-set(p3) = {i3}) | possible-type(i3) \sqsubseteq possible-type(i1) possible-type(*p3) \sqsubseteq possible-type(i3) |

| Final possible-types: | required-types: |
|-------------------------------|-------------------------------|
| possible-type(i1) = int | required-type(i1) = int |
| possible-type(i2) = int | required-type(i2) = int |
| possible-type(f2) = \perp | required-type(f2) = float |
| possible-type(i3) = int | required-type(i3) = int |
| possible-type(p1) = valid-ptr | required-type(p1) = valid-ptr |
| possible-type(p2) = valid-ptr | required-type(p2) = valid-ptr |
| possible-type(p3) = pointer | required-type(p3) = valid-ptr |
| possible-type(*p1) = int | required-type(*p1) = int |
| possible-type(*p2) = \perp | required-type(*p2) = int |
| possible-type(*p3) = int | required-type(*p3) = int |

Figure 10.4 Computing *possible-type* for the example in Figure 10.1.

Rule L1 recognizes dereferences that may violate memory safety: if a pointer's *required-type* is *valid-ptr* (which can arise only if the pointer is dereferenced; see Rule R2), and its *possible-type* is not *valid-ptr*, then it may violate memory safety. Rule L2 classifies as *type-unsafe* any expression whose *possible-type* is not compatible with its *required-type* (note that in the lattice of types, *valid-ptr* is compatible with *pointer*, and *init* is compatible with all scalar types). Any expression that is not classified as *mem-unsafe* or *type-unsafe* by rule L1 or L2 is *safe*.

After classifying the *lvalue* expressions, we classify the locations. Recall that locations comprise variables, which have a corresponding *lvalue* expression, and MALLOC and STRLIT objects, which do not (they may only be accessed indirectly via a dereference). For each location x :

- L4. If x is a variable, and the *lvalue* expression x is *type-unsafe* or *mem-unsafe*, then the location x is *unsafe*.
- L5. If x is not a variable, and there is some \mathbf{p} such that $x \in pt\text{-set}(\mathbf{p})$ and $possible\text{-type}(x) \not\sqsubseteq required\text{-type}(*\mathbf{p})$, then the location x is *unsafe*.
- L6. If x has not been classified as *unsafe* (by Rule L4 or L5), and there is some \mathbf{p} such that $x \in pt\text{-set}(\mathbf{p})$ and $*\mathbf{p}$ is *type-unsafe* or *mem-unsafe*, then the location x is *tracked*.
- L7. Otherwise, x is *untracked*.

Rule L4 simply says if a location is also an *lvalue* expression, and that *lvalue* expression has been classified as *type-unsafe* or *mem-unsafe*, then that location is *unsafe*. Rule L5 states that if the *possible-type* of a MALLOC or STRLIT location is not compatible with the *expected-type* of some dereference that could access it, then that location is also *unsafe*. Rule L6 classifies as *tracked* any variable in the points-to set of a pointer \mathbf{p} whose dereference expression $*\mathbf{p}$ is either *type-unsafe* or *mem-unsafe*. Any location not classified as *unsafe* or *tracked* by rules L4–L6 is *untracked*.

| Code | RTC Instrumentation |
|--|---|
| <pre> int i1; int * p1; int i2; float f2; int * p2; int i3; int * p3; </pre> | <pre> SetTag(&i1, uninit_type); SetTag(&p1, uninit_type); SetTag(&i2, uninit_type int_type); SetTag(&f2, uninit_type); SetTag(&p2, uninit_type); SetTag(&i3, uninit_type int_type); SetTag(&p3, uninit_type); </pre> |
| <pre> 1. p1 = &i1; 2. *p1 = 23; </pre> | <pre> SetTag(&p1, pointer_type); VerifyType_use(&p1, pointer_type); VerifyDeref(&*p1, int_type); SetTag(&*p1, int_type); </pre> |
| <pre> 3. i2 = *p1; </pre> | <pre> VerifyType_use(&p1, pointer_type); VerifyType_assign(&*p1, int_type); CopyTag(&i2, &*p1, sizeof(int)); </pre> |
| <pre> 4. f2 = 6.7; 5. if(*p1 == 0) </pre> | <pre> SetTag(&f2, float_type); VerifyType_use(&p1, pointer_type); VerifyType_use(&*p1, int_type); </pre> |
| <pre> 6. p2 = &i2; 7. else 8. p2 = (int *) &f2; 9. *p2 = 4; </pre> | <pre> SetTag(&p2, pointer_type); VerifyType_use(&p2, pointer_type); VerifyDeref(&*p2, int_type); SetTag(&*p2, int_type); </pre> |
| <pre> 10. p3 = &i3 + 1; 11. *p3 = i1; </pre> | <pre> SetTag(&p3, pointer_type); VerifyType_assign(&i1, int_type); VerifyType_use(&p3, pointer_type); VerifyDeref(&*p3, int_type); CopyTag(&*p3, &i1, sizeof(int)); SetTag(&*p3, int_type); </pre> |

Figure 10.5 RTC instrumentation removed with Type-Flow Analysis.

Looking back at the example in Figure 10.4, Rule L1 makes `*p3` *mem-unsafe*, Rule L2 makes `f2` and `*p2` *type-unsafe*, and Rule L4 makes `i2` and `i3` *tracked*. All remaining *lvalue* expressions are *safe*, and all remaining locations are *untracked*.

Figure 10.5 shows the RTC instrumentation for the Figure 10.4 example, crossing out instrumentation that is eliminated as a result of using Type-Flow Analysis. For the *tracked* locations `i2` and `i3`, when declared, their tags are set to their possible type (*int*) rather than *uninit*. At line 11, the `CopyTag` call is replaced with a `SetTag` call, because the right-hand-side of the assignment is a *safe* expression (`i1`) with a known static type. As can be seen, the analysis eliminates a significant portion of the runtime instrumentation.

Note that *untracked* locations will always be tagged *unalloc* in the mirror. If location x is accessed indirectly by an invalid dereference `*p`, the dereference expression `*p` will have been classified as *mem-unsafe*, and thus will be instrumented with a `VerifyDeref` check. Since x is tagged *unalloc*, the check of `*p` will trigger a memory-safety error. Thus, as was the case with the static analysis in the MSE, the type-flow analysis increases the likelihood of the RTC tool detecting a memory-safety error.

10.1.5 Experimental Results

Figure 10.6 gives the runtime slowdown of RTC with and without the use of type-flow analysis. The improvements are significant, reducing the average slowdown from $44\times$ down to $27\times$ (a 38% improvement). Figure 10.7 shows the compilation-time slowdown with and without type-flow analysis: on average, the analysis only contributed an additional $1\times$ slowdown. In most cases the analysis time is negligible, while in the larger benchmarks, the analysis time is more significant, though still quite low.

10.2 Stack and Heap Locations

The type-flow analysis, as described, unsafely assumes that temporal access errors do not occur. That is, a dereference that may access a location after it has been deallocated may still be classified as *safe*. To classify such a dereference as *mem-unsafe*, we make the

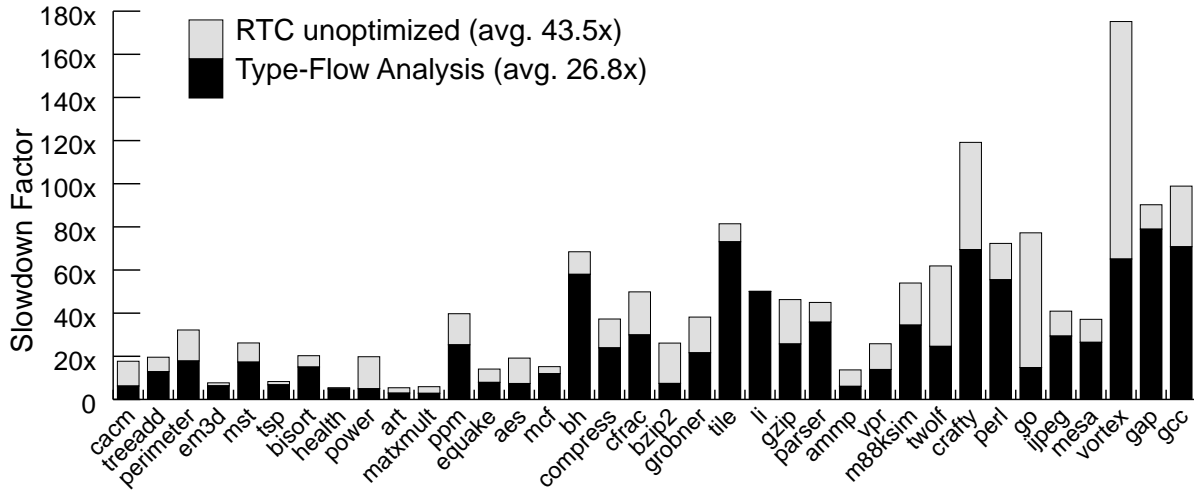


Figure 10.6 RTC Runtime Overhead: Type-Flow Analysis

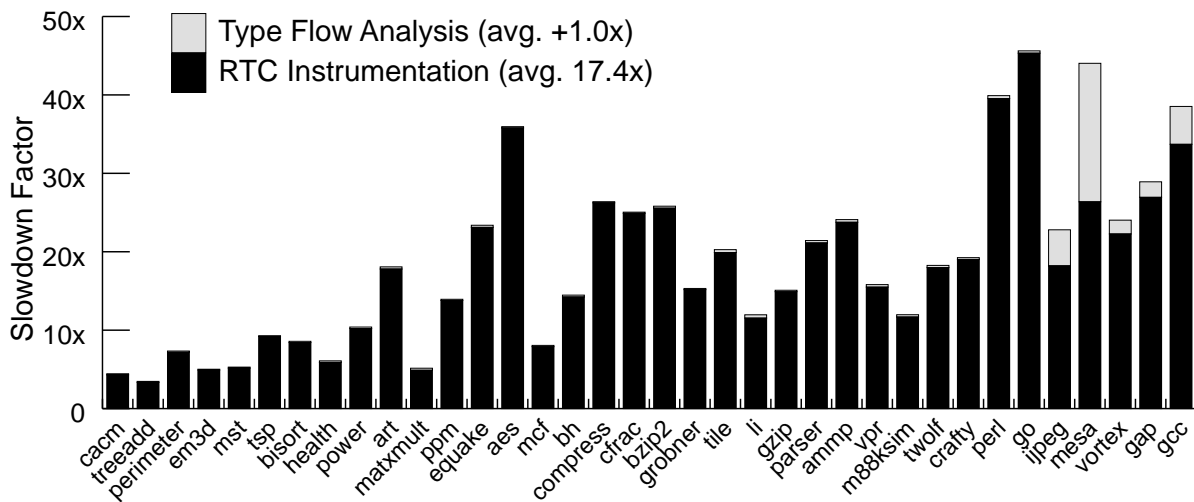


Figure 10.7 RTC Analysis Time: Type-Flow Analysis

conservative assumption that any dereference that may access a stack location or a heap location that is ever freed could potentially cause a temporal access error (this is the same assumption made in the EPT analysis for the MSE, in Section 4.2).

This requires the following change to the type-flow analysis: the expression $\&x$ has type *valid_ptr* only if x is a global or static variable, while $\&MALLOC_i$ has type *valid_ptr* only if

`MALLOCi` is never freed. Otherwise, the expression has type *pointer*. The following example illustrates the difference:

| Program | Assignment Edges | Possible Types |
|--|--|--|
| 1. <code>static int s;</code> 2. <code>auto int a;</code> 3. <code>int *p1 = &s;</code> 4. <code>int *p2 = &a;</code> 5. <code>*p1 = *p2;</code> | $p1 \stackrel{=}{\leftarrow} \text{VALUE}_{\text{valid-ptr}}$ $p2 \stackrel{=}{\leftarrow} \text{VALUE}_{\text{pointer}}$ | $\text{possible-type}(p1) = \text{valid-ptr}$ $\text{possible-type}(p2) = \text{pointer}$ |

Since both `p1` and `p2` are dereferenced (at line 5), the *required-type* of both are *valid-ptr*; thus, Rule L1 classifies `*p2` as *mem-unsafe*, while `*p1` can be classified as *safe*.

Figure 10.8 shows the runtime slowdown of RTC with type-flow analysis, comparing the “unsafe” version that assumes temporal access errors never occur with the “safe” version that assumes any pointer to stack or freed heap may potentially cause a temporal access error. The difference is small (introducing a 2% slowdown on average), with most of the difference coming from parameters passed by reference (as was the case with the MSE, Section 4.3).

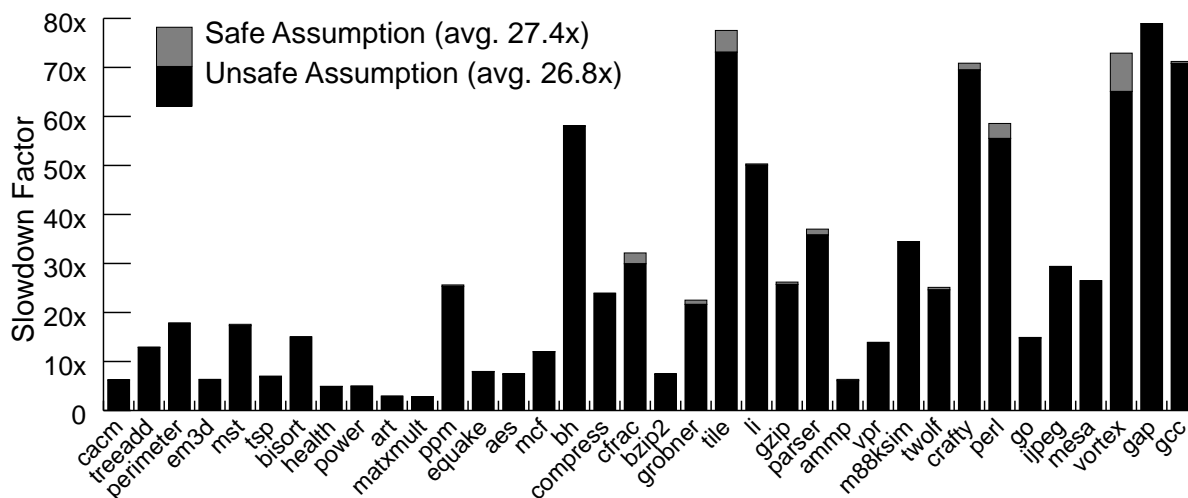


Figure 10.8 RTC Runtime Overhead: Heap and Stack Assumptions

10.3 May-Be-Uninitialized Analysis

Another unsafe assumption made by the type-flow analysis is that uninitialized memory access errors never occur. By eliminating all instrumentation for *safe* expressions and *untracked* locations, and by initializing *tracked* locations to their statically-determined types rather than to *uninit*, the RTC tool would no longer detect uninitialized memory accesses in these locations (though it would still detect uninitialized memory accesses to *unsafe* locations). To address this problem, a flow-sensitive analysis is needed to find program points where instrumentation cannot be elided.

For a location x that is *untracked* or *tracked* (i.e., is not unsafe), the analysis finds instances of x where x may be uninitialized. This analysis is defined as a forward intra-procedural dataflow-analysis problem:

- The elements of the underlying lattice are sets of locations.¹
- The analysis computes two sets for each CFG node n : $Uninit_{in}^n$ and $Uninit_{out}^n$, representing locations that *may* contain uninitialized data before and after n .
- The dataflow functions are as follows:
 - At a node n that declares a stack variable x without initializing it, or that allocates the heap location x (excluding via `calloc`, which zero-initializes the allocated memory), $Uninit_{out}^n = Uninit_{in}^n \cup \{x\}$
 - At a node n that is a direct assignment $\mathbf{x} = e$ (where \mathbf{x} is a variable),
 - ▶ if e is a variable y such that $y \in Uninit_{in}^n$, then

$$Uninit_{out}^n = Uninit_{in}^n \cup \{\mathbf{x}\}$$
 - ▶ if e is a dereference $*\mathbf{q}$ such that $pt\text{-}set(\mathbf{q}) \cap Uninit_{in}^n \neq \emptyset$, then

$$Uninit_{out}^n = Uninit_{in}^n \cup \{\mathbf{x}\}$$
 - ▶ if e is a call to function \mathbf{f} , then

$$Uninit_{out}^n = Uninit_{in}^n \cup MayMod(\mathbf{f}) \cup \{\mathbf{x}\}$$

¹This can be extended to include structure fields; we omit this detail in this discussion.

where $MayMod(\mathbf{f})$ is the set of locations that *may* be modified as a result of calling the function. Essentially, a call to \mathbf{f} is treated as possibly assigning an *uninit* value to all locations in $MayMod(\mathbf{f})$.

► otherwise, if \mathbf{x} is a scalar variable, then

$$Uninit_{out}^n = Uninit_{in}^n - \{\mathbf{x}\}$$

► otherwise, $Uninit_{out}^n = Uninit_{in}^n$

■ If node n contains an indirect assignment $*\mathbf{p} = e$,

► if e is a variable \mathbf{y} such that $\mathbf{y} \in Uninit_{in}^n$, then

$$Uninit_{out}^n = Uninit_{in}^n \cup pt\text{-set}(\mathbf{p})$$

► if e is a dereference $*\mathbf{q}$ such that $pt\text{-set}(\mathbf{q}) \cap Uninit_{in}^n \neq \emptyset$, then

$$Uninit_{out}^n = Uninit_{in}^n \cup pt\text{-set}(\mathbf{p})$$

► if e is a call to function \mathbf{f} , then

$$Uninit_{out}^n = Uninit_{in}^n \cup MayMod(\mathbf{f}) \cup pt\text{-set}(\mathbf{p})$$

► otherwise, $Uninit_{out}^n = Uninit_{in}^n$

• The lattice meet is set union.

After performing the may-be-uninitialized analysis, instrumentation is added to allow the RTC tool to detect uses of uninitialized data. A *tracked* or *untracked* location x for which there is a use of x in a node n where $x \in Uninit_{in}^n$ is treated as follows:

1. The declaration of x is instrumented to set its tag to *uninit*.
2. For each node n that uses x , if $x \in Uninit_{in}^n$, then the use of x must be instrumented (with a call to `VerifyType_use`).
3. For each node n that assigns to x , if either $x \in Uninit_{in}^n$ or $x \in Uninit_{out}^n$, then the assignment must be instrumented (with a call to `CopyTag` or `SetTag`). This ensures that x 's tag is set correctly for subsequent uses of x . The only assignments to x that need not be instrumented are those for which x is definitely not uninitialized both before and after the definition.

Additionally, for each indirect assignment `*p = ...` that is instrumented as a result of condition 3, each location in `p`'s points-to set that was previously classified as *untracked* must now be considered *tracked*, so that its RTC mirror will not be tagged *unalloc* at runtime.

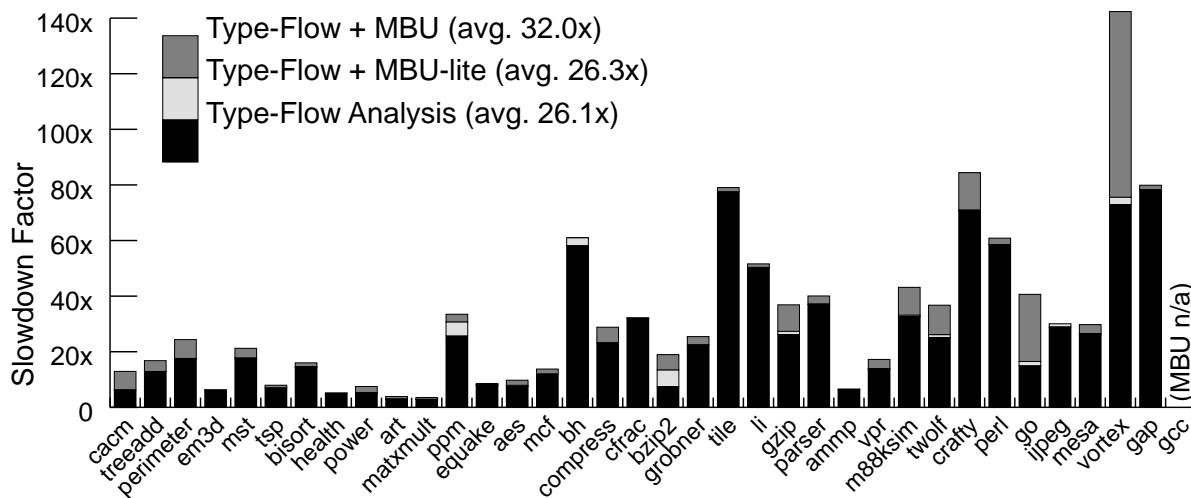


Figure 10.9 RTC Runtime Overhead: May-Be-Uninitialized Analysis

Figure 10.9 shows the runtime overhead after instrumentation is reintroduced for locations and expressions that may be uninitialized (MBU) compared to the runtime overhead of type-flow analysis. (‘MBU-lite’ is explained later.) The analysis did not complete for `gcc`, so its numbers are excluded from the averages. The performance has been significantly degraded, from $26\times$ (with the ‘unsafe’ Type-Flow analysis) to $32\times$. Worse yet, Figure 10.10 gives the analysis times for may-be-uninitialized analysis, which is very slow for the larger benchmarks.

One big reason for the inefficiency and imprecision of may-be-uninitialized analysis is the need to account for propagating the *uninit* type along assignments. In particular, for a weak update to `*p`, if the right-hand-side *may* be uninitialized, then any location in `p`'s points-to set must be considered possibly uninitialized. With imprecisions due to points-to analysis, and imprecise handling of structures and function calls, this causes the size of the *Uninit* facts to explode quickly. (We also tried performing the may-be-uninitialized analysis

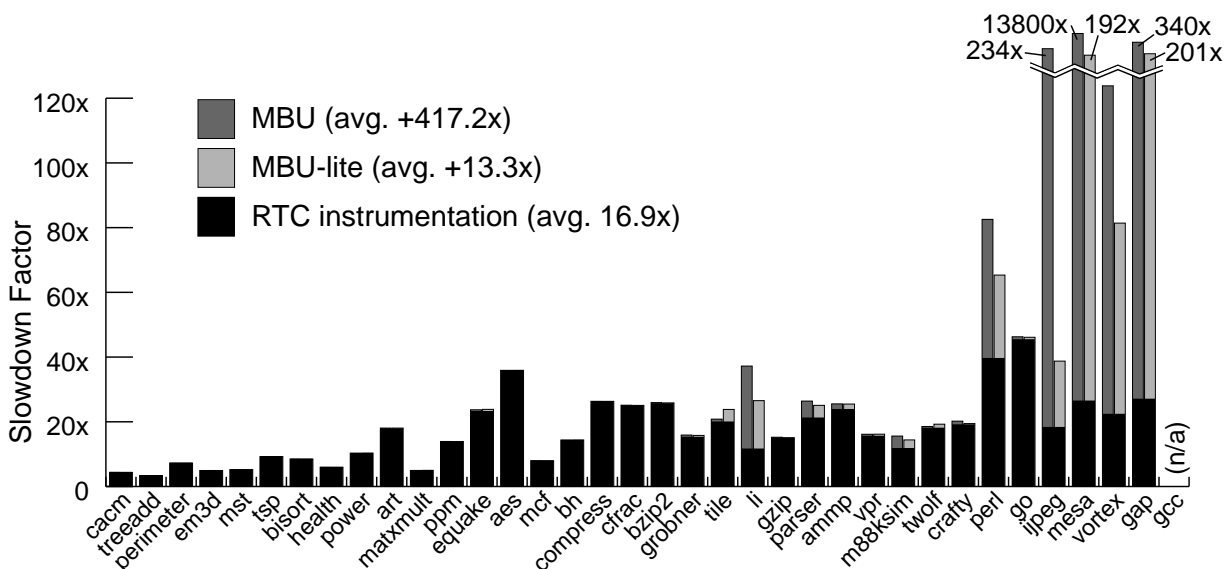


Figure 10.10 RTC Analysis Time: May-Be-Uninitialized Analysis

inter-procedurally in a context-insensitive manner, and the analysis time was even slower, with only marginal improvements in runtime overhead.)

One solution to this problem, which we call MBU-lite, is to change the runtime behavior of the RTC so that an assignment of an *uninit* value causes the destination to be tagged with the static type of the assignment (after reporting a warning). This allows the dataflow analysis to be more efficient, since assignments no longer add new locations to the *Uninit* dataflow fact. This change would only weaken the effectiveness of the RTC as a diagnostic tool, not as an error-reporting tool: an erroneous use of an uninitialized value may not be reported at the point of use, but it will have been reported as a warning at an earlier point in execution, when the uninitialized value is first assigned. The execution time and analysis time slowdowns with MBU-lite are shown in Figures 10.9 and 10.10: the runtime overhead introduced by MBU-lite is negligible compared to the overhead of RTC with type-flow analysis, and the analysis time overhead is a significant improvement over MBU, though there are still some scalability issues with the larger programs (*gcc* still did not complete).

This situation presents some tradeoffs between runtime performance, analysis efficiency, and coverage:

1. RTC unoptimized: catches uninitialized memory access errors, $17\times$ compile time slowdown, $42\times$ execution time slowdown.
2. Type-Flow analysis only: does not catch uninitialized memory access errors, negligible compile time slowdown, $26\times$ execution time slowdown.
3. Type-Flow + MBU analysis: catches uninitialized memory access errors, compile time not scalable (up to $13800\times$ slowdown), $32\times$ execution time slowdown.
4. Type-Flow + MBU-lite: reports uninitialized memory assignments as warnings, compile time not scalable, $26\times$ execution time slowdown.

Due to the scalability problem, options 1 or 2 may be preferable to options 3 or 4 for larger programs.

10.4 Redundant Checks Analysis

When an *lvalue* expression e is *used* many times with no intervening change to e 's *lvalue* or runtime type, a runtime check is only necessary for the first *use* of e . This is because if the first check of e reports a runtime error, the RTC tool will set the runtime type of e 's location to its statically declared type to prevent cascading errors. A subsequent check of e , with no intervening change to the *lvalue* of e or to the runtime type of e , is therefore redundant, and can be eliminated.

To identify redundant checks, we perform an intra-procedural dataflow analysis to keep track of *lvalue* expressions that have been checked. A check of an unsafe expression e is redundant at control-flow graph node n if every path from each function's *enter* node to n includes a node m such that:

- There is a use of e at node m (where e will be instrumented by the RTC tool to check its runtime type), and

- No path from m to n changes the runtime type of e , changes the *lvalue* of e , or deallocates a location that may be accessed by e .

The analysis is defined as follows:

- The elements of the underlying lattice are sets of type-unsafe and mem-unsafe *lvalue* expressions (either a variable v or a pointer dereference $*p$).
- The analysis computes two sets for each CFG node n : $Checked_{in}^n$ and $Checked_{out}^n$, representing expressions for which checks would be redundant before and after n .
- For each function's enter node n_0 , $Checked_{in}^{n_0} = \emptyset$
- The lattice meet is set intersection.
- The dataflow function for each node n is of the form

$$Checked_{out}^n = Checked_{in}^n \cup Gen(n) - Kill(n)$$

where $e \in Gen(n)$ if e is type-unsafe or mem-unsafe and there is a use of e at node n , and $e \in Kill(n)$ if n changes the *lvalue* or runtime-type of e (see below). Note again the difference between this dataflow function and the conventional *Gen/Kill* problems, where the *Kill* set is removed before the *Gen* set is added.

We split the *Kill* set into two components: $Kill(n) = Kill_{lval}(n) \cup Kill_{type}(n)$, where $Kill_{lval}(n)$ includes all expressions whose *lvalue* may be changed or deallocated by n , and $Kill_{type}(n)$ includes all expressions whose runtime type may be changed by n . The $Kill_{lval}(n)$ set is identical to the *Kill* set described in Chapter 5 for the MSE. The $Kill_{type}(n)$ set is defined as follows.

An *lvalue* expression e is in $Kill_{type}(n)$ if

1. n contains an assignment $e_1 = e_2$, where e_2 is an *lvalue* expression (a variable or a dereference) that is type-unsafe or mem-unsafe, and $alias-locs(e) \cap alias-locs(e_1) \neq \emptyset$. (The set $alias-locs(e)$, defined in Chapter 5, is the set of locations that may be validly accessed by the expression e .)

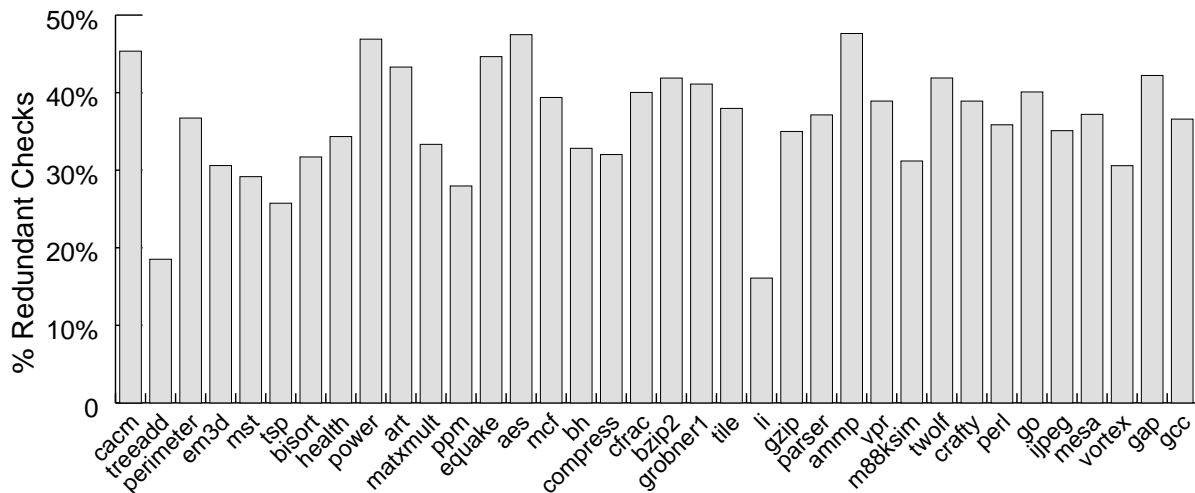


Figure 10.11 RTC Redundant Checks

2. n contains a call to f , and $alias-locs(e) \cap MayMod(f) \neq \emptyset$.

After performing this analysis, a use of an expression e at a node n need not be instrumented if e is in $Checked_{in}^n$. Figure 10.11 gives the percentage of RTC checks that were found to be redundant, and thus removed: on average 36.3% of instrumentation was removed.

Figure 10.12 shows the runtime overhead of the RTC with type-flow analysis ($27\times$), and with both type-flow and redundant checks analysis ($22\times$). Redundant checks analysis improved the overhead by 19% on average, with bigger improvements in the larger benchmarks. However, the cost in terms of analysis time can be quite high. Figure 10.13 shows the compilation time slowdown introduced by both type-flow analysis and redundant checks analysis — with the latter contributing an additional $4.8\times$ slowdown on average, but significantly more for some of the larger benchmarks.

10.5 Summary of RTC Optimizations

We have described two improvements to the runtime performance of the RTC. The first uses a flow-insensitive type-flow analysis to determine which expressions are guaranteed never to violate type safety or memory safety. This approach is scalable, and improves the

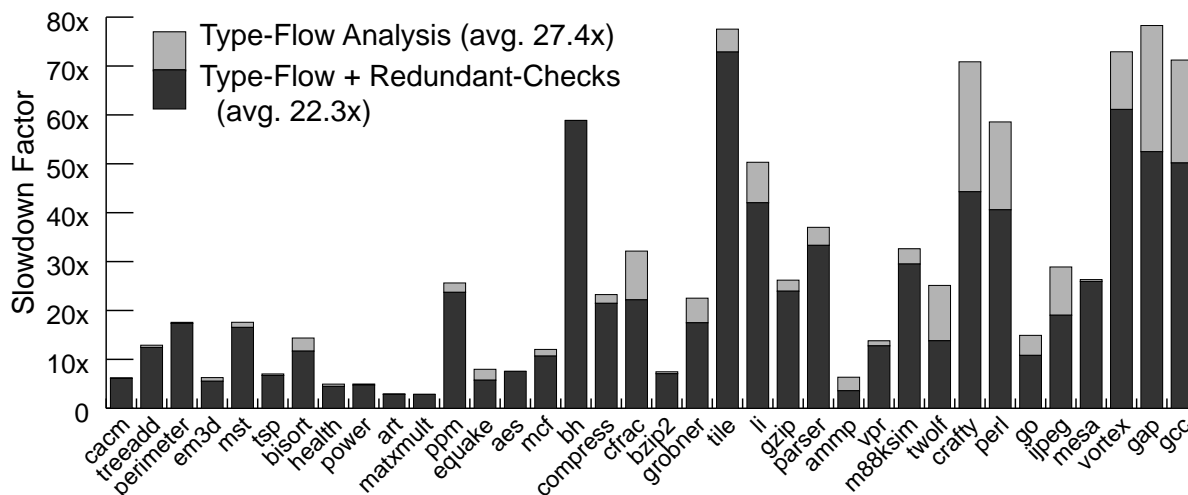


Figure 10.12 RTC Runtime Overhead: Redundant Checks Analysis

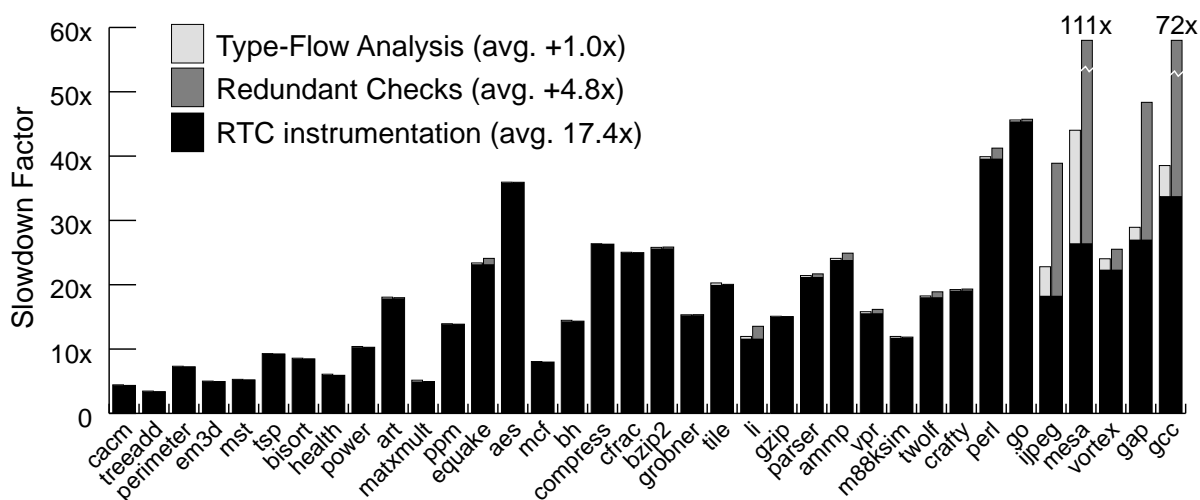


Figure 10.13 RTC Analysis Times: Redundant Checks Analysis

runtime overhead significantly (from $44\times$ to $27\times$). However, it does not account for possible uses of uninitialized memory. The flow-sensitive may-be-uninitialized analysis addresses this weakness, but for the larger programs, the runtime overhead improvements gained by type-flow analysis is degraded, and the analysis does not scale.

The second improvement to the RTC uses the flow-sensitive redundant checks analysis, which gives reasonable runtime overhead improvements (from $27\times$ to $22\times$), but does not scale well to large programs.

Chapter 11

Related Work

In this chapter, we describe work related to our runtime monitoring approach for preventing security violations and for finding programming errors. We organize our survey of related work by first exploring runtime monitoring approaches in increasing order of refinement, beginning with security-oriented approaches that only target known vulnerabilities, progressing into approaches that enforce memory safety and type safety. We then touch upon some approaches that supplement dynamic checks with static analysis, and error-detection approaches that are purely static.

11.1 Security-Oriented Approaches

With software security becoming a growing concern, a number of runtime-monitoring approaches have been developed with the goal of preventing known methods of attacks from succeeding. Most of these approaches have focused on buffer overruns, which is the most common vulnerability to be found in programs, and stack smashing, which is probably the easiest way to exploit a buffer-overrun vulnerability.

StackGuard [Cow⁺98, Cow⁺00] is a security tool that instruments a program to place a “canary” word next to the return address in the activation record when entering a new function. Prior to returning from the function, the canary value is checked to see if it has changed; if so, the return address may have been tampered with. PointGuard [Cow⁺00] is an extension of StackGuard that uses the same canary approach to detect tampering of function

pointers and `longjmp` buffers. The approach has a low runtime overhead, but the adjacent-canary approach assumes that the mechanism for overwriting the vulnerable location is a “continuous overflow” that overwrites a contiguous block of memory; the protection can be circumvented by a “direct write” method — e.g., via a format-string vulnerability — to modify the return address without changing the canary value.

To address this shortcoming, they tried using a fine-grained memory-protection mechanism called MemGuard [Cow⁺98] to write-protect the return address in the activation record; however, the overhead incurred was exorbitant. Other approaches for protecting the return address have also addressed this concern. StackShield [Sta00] copies the return address to an array, the Global Ret Stack, declared elsewhere in memory; the return value is retrieved from the Global Ret Stack prior to returning from a function. This approach effectively shifts the sensitive ‘return-address’ location from the activation record to the Global Ret Stack, which is still prone to tampering — i.e., an attack could be effected by overwriting the Global Ret Stack rather than the activation record. The Return Address Defender (RAD) [Chi⁺01] addresses this shortcoming by protecting the memory containing the return addresses — in their case, called the Return Address Repository (RAR) — in one of two ways:

1. *MineZone RAD* surrounds the RAR with buffers that are declared read-only. This prevents the RAR from being written by a continuous overflow, but not by a direct write.
2. *Read-Only RAD* declares the RAR itself to be read-only, thus preventing it from being corrupted via direct writes. However, this introduces a significant runtime overhead, since the RAR needs to be un-protected and re-protected at the start of each function to push the return address onto the RAR.

To achieve both better protection and performance, SmashGuard [Özd⁺02] proposes storing the return addresses in a hardware stack.

Libverify and Libsafe [Bar⁺00] are libraries that respectively check for stack smashing and buffer overruns. Libverify implements the StackGuard canary approach by adding instrumentation at link time rather than compile time. Libsafe contains instrumented versions of library functions that are normally prone to buffer overruns (like `strcpy`). The libsafe versions of these functions check that writes to a buffer do not exceed the bounds of the stack frame containing the buffer. This prevents library functions from overflowing into the return address, but does not prevent them from overflowing into other sensitive locations within the same stack frame, like function pointers, nor does it prevent overflows that are not caused by a library function.

These approaches are, by and large, *ad hoc* solutions to security vulnerabilities, and only address specific forms of attack that are known and well-understood. Because most attacks take advantage of weaknesses in the C language and its weak enforcement of memory and type safety, there is a trend towards developing stronger versions of C to prevent *any* erroneous behavior from happening.

11.2 Memory and Type Safety

The lack of memory-safety enforcement is arguably the weakest aspect of the C language: memory-safety errors are easy to induce, can be difficult to detect and diagnose, and are most vulnerable to attacks. For this reason, memory safety has been the focus of many approaches to finding errors and preventing attacks.

11.2.1 Fat Pointers

Fat pointers have been used to check for memory safety violations in `bcc` [Ken83], Integral-C [Ros86], Saber-C [Kau⁺88], and `rtcc` [Ste92]. These were all designed for use in a debugging setting, and incurred high runtime overheads. Further, they only detected spatial access errors and not temporal access errors.

Safe-C [Aus⁺94] was the first approach to systematically account for temporal access errors. This was done by associating each storage allocation (stack frame or `malloc` block)

with a unique ID (called a “capability”), and augmenting the fat pointer to record the capability of the intended target. They also used a runtime optimization to skip redundant checks, and had a 2–6× runtime overhead. Keen *et al.* [Kee⁺02a] proposed using hardware to improve this performance, with a hardware accelerated table to speed up the temporal checks, and a hardware reduction queue to detect redundant checks dynamically.

Guarding [Pat⁺97] is an approach that performs spatial and temporal access checks similar to Safe-C, but it decouples the access checks from the original program, and executes these checks in a *shadow process* separately from the original program — possibly in parallel on a multi-processor, to improve runtime performance.

Cyclone [Jim⁺02] is version of the C language that enforces memory-safety at runtime. The language distinguishes between *never-NULL* pointers, regular pointers, and fat pointers, and instruments uses of the latter two with runtime checks for null dereference and spatial access errors respectively. To prevent temporal access errors, Cyclone requires the use of a garbage collector or a region-based memory-management scheme [Gro⁺02]. Due to the need to convert C pointers into the appropriate kind of Cyclone pointer, and to adapt to the region-based memory management, porting existing C programs into Cyclone becomes a significant barrier to its widespread use.

CCured [Nec⁺02, Con⁺03] is a system that enforces memory safety and limited type safety (distinguishing only between pointers and non-pointers) in C programs. It uses fat pointers to detect spatial access errors, and garbage collection to prevent temporal access errors. To make this approach efficient, it restricts some of the flexibility of C, and so disallows certain valid C operations (like storing the address of a stack variable on the heap, and dereferencing a pointer that has been cast to/from an integer). It may also reports some false positives, such as when accessing open arrays declared with size 1 (see Section 3.1, page 26), or assigning between pointers declared to point to different-sized underlying array types. To reduce the overhead of runtime checks, CCured uses a static type-inference scheme to classify pointers into *safe*, *sequence*, and *dynamic* pointers, so that runtime checks can be reduced for sequence pointers and eliminated for safe pointers. The goal of their type

inference is thus similar to that of our Extended Points-To and Type-Flow analyses, and the inference was able to improve CCured’s runtime overhead from 6–20× to 1–2× [Nec⁺02].

A major complication with the fat-pointer approaches is in interacting with uninstrumented libraries. Due to the change in the pointer representation, fat pointers must be translated to regular pointers before being passed to uninstrumented functions — an exercise that requires manual effort [Har⁺03]. To address this concern, Jones and Kelly [Jon⁺97] enforce memory safety by storing the set of valid pointer targets in a separate data structure (in their implementation, a splay tree), and check each pointer operation (including pointer arithmetic) to ensure that it remains within the bounds of its valid target. This allows the pointer representation to remain unchanged, which lets instrumented modules interoperate with uninstrumented ones, but their runtime overhead is high (5–6× slowdown), and they restrict certain unorthodox operations like computing an intermediate pointer value more than one byte past the end of an array.

CRED [Ruw⁺04] extends the Jones and Kelly approach to allow pointers to point beyond one past the end of an array. When a pointer value is computed that points beyond one past the end of an array, an additional level of indirection is introduced to associate the pointer with a reference to its intended target. While this enables CRED to handle a larger set of programs than Jones and Kelly, there remain cases that are not safely handled by CRED, such as dereferencing a pointer value cast from an integer that has undergone integer arithmetic. The overhead of CRED is similar to Jones and Kelly, and quite high (up to 12× slowdown), but restricting their approach to check only `char` pointer dereferences was found to decrease runtime overhead significantly (to 1.3–2.3× slowdown). They argue that since most buffer-overflow attacks exploit string operations (that operate via `char` pointers), restricting the checks to `char`-pointer operations does not reduce the protection of CRED.

Compared to the Jones and Kelly approach and CRED, our tagged memory approach has the benefit of interoperability with uninstrumented modules while maintaining a low runtime overhead, without restricting the language, and without limiting checks to `char`

pointers. In fact, the `char`-pointer-only checking is an option that could be adopted by the MSE to further reduce our runtime overhead.

11.2.2 Tagged Memory

The tagged memory approach sacrifices completeness for efficiency and practicality: while it cannot guarantee detecting *all* memory safety errors, it can be much more flexible than the fat-pointers approach while still being able to detect *most* errors. It has been effectively employed in a number of tools.

Purify [Has⁺92], is a dynamic debugging tool for detecting memory-safety errors, memory leaks, and uninitialized memory accesses. It tags memory with two bits for each byte of memory: one bit to indicate whether the byte is allocated, and one bit to indicate whether the byte is initialized. It instruments object code, and has a fairly high overhead (about 15× slowdown) which is acceptable for a debugging tool, but impractical for use in deployed code.

Valgrind [Net⁺03] performs similar checks, but it interprets the executable binary on a “synthetic CPU”, and thus incurs a higher overhead (about 40× slowdown). The `memcheck` component of Valgrind associates each byte of memory with one *valid-address* bit to mark allocated memory and eight *valid-value* bits to detect uses of uninitialized memory.

Hobbes [Bur⁺03] is another interpreter for binary code; it includes runtime type-checking that is very similar to that performed by the RTC. Each byte of memory is shadowed by a one-byte tag, which encodes its scalar type along with an *initialized* flag, and an *invariant* flag indicating whether the type of the byte is permitted to change. The interpreter incurs a 40× slowdown, and the type-checker an additional 100× slowdown.

These approaches all operate on binary code, which gives them the advantage of not requiring source code, but a disadvantage of being platform dependent, and being unable to take advantage of source-level information to improve runtime performance or coverage with static analysis. In our source-level approach, we were able to use static analysis to achieve significantly better runtime overheads.

11.2.3 Runtime Type Checking

Besides the RTC and Hobbes, there have been few other attempts to check type safety in C programs at runtime. The Saber-C [Kau⁺88] programming environment includes a rich interpreter-based debugging facility that, among other things, checks for both memory safety and type safety. Fail-Safe C [Oiw⁺02] is an approach to fully enforce memory and type safety at runtime for the full ANSI C. Memory safety is checked using fat pointers, while type safety is checked by maintaining data structures describing each (aggregate) type declared in the program, and using those data structures to resolve dereferences that have been affected by a type cast. Due to the change in data representation, they, like CCured, also require a complicated mechanism for interfacing with uninstrumented libraries, which also introduces some runtime overhead [Sue⁺03].

Runtime type checking is more prevalent in functional-style languages, and in languages with higher-order type systems. Despite the significant differences in the language structure and type system between these languages and C, there is some related work that is relevant to the RTC, in particular to optimizing runtime performance. For dynamically-typed languages like LISP and Scheme, Henglein [Hen92] proposed an efficient type-inference algorithm very similar in spirit to our flow-insensitive Type-Flow Analysis. Soft Typing [Car⁺91] is a combination of static and dynamic typing, where a program is first statically type-checked as much as possible, before dynamic type-checking is applied to the parts of the program that did not pass the static type-checker. Chailloux *et al.* [Cha⁺97] argued that, although runtime tags are not necessary for ML programs to run correctly, they can be useful for debugging, as well as to aid the garbage collector. They described an approach that stores type information for the heap in a mirror space, which allowed their implementation of a stop-and-copy garbage collector to be space efficient.

11.2.4 Other Runtime Monitoring Ideas

DynamoRIO [Dyn] is an approach to add runtime-monitoring checks on-the-fly (i.e., while the program is running), and to use dynamic optimization techniques to improve

performance. *Program shepherding* [Kir⁺02] is a specific application built on top of this infrastructure for enforcing security policies at runtime; this approach is able to guarantee that a given runtime check can never be circumvented.

There has been some recent work that extends dynamic approaches to detect potential errors that are not necessarily exercised during execution. Haugh and Bishop [Hau⁺03] focus on library functions that are known to be vulnerable to buffer overruns, like `strcpy`, and report a potential error if, for example, the size of the source buffer in a call to `strcpy` is greater than the size of the destination buffer, even though no buffer overrun actually occurs on the given execution. Larson and Austin [Lar⁺03] describe a similar but more general and more precise approach. They track more runtime information (such as the maximum potential string length in a given buffer) and handle more operations (beyond the vulnerable library functions), allowing them to detect more errors and fewer false positives.

11.3 Eliminating Array-Bounds Checks

Another relevant area that has been extensively researched is the elimination of runtime array-range checks. Early work includes Harrison [Har77], who relied on the presence of structured loops to infer range information, and Welsh [Wel78], who made use of explicit subrange type declarations in Pascal to optimize range checks. Markstein *et al.* [Mar⁺82] described using code motion and common subexpression elimination to optimize range checks, approaches later refined by Asuru [Asu92], Gupta [Gup93], and Kolte and Wolfe [Kol⁺95]. Bodik *et al.* [Bod⁺00] described an efficient analysis that allows array-bounds checks in Java programs to be eliminated on-demand. Luján *et al.* [Luj⁺02] explore ways to allow bounds-checking elimination strategies to be effective in the multi-threaded setting of Java. Most of the approaches studied have dealt with languages in which array accesses are more tightly controlled, while our pointer-range analysis had to contend with complications arising from pointer arithmetic and type casts in C.

11.4 Static Error Detection

The goal of the SAFECODE [Kow⁺02, Dhu⁺03] project is to statically validate that memory safety errors can never occur, thus avoiding the need for runtime checks. In essence, the idea amounts to restricting the set of valid programs to be those that pass their static checks. An example restriction is that array indices must be affine in relation to the array sizes; this property is required because of the limitations of range analysis, but is non-trivial to state, and might not be easy for a programmer to adhere to.

Static error-detection techniques analyze a program without executing it to find potential errors or security holes. The main benefit of static techniques is that they can detect errors in portions of the program that are infrequently executed. Unfortunately, precise static techniques are expensive, and thus do not scale to large programs. In order for static approaches to scale, either the user must supply annotations (as in LCLint/Splint [Eva96, Lar⁺01] and ESC [Det⁺98]), or a less precise analysis must be performed (which may lead to missing some potential errors or to reporting false positives [Wag⁺00, Dor⁺01]), or the scope of the analysis must be limited, either by checking only certain paths (like PREFIX [Bus⁺00]), or simplifying the properties to be checked [Bal⁺01, Ash⁺02, Liv⁺03, Che⁺04].

Chapter 12

Conclusion and Future Directions

This thesis has described a framework for monitoring C programs at runtime to prevent security violations and to detect programming errors. The approach can be used to check properties at varying levels of refinement. Three manifestations of our framework have been explored, with different design goals and runtime overheads:

SLC (Sensitive Location Checker): a security-oriented approach that prevents erroneous writes from corrupting specific sensitive locations (like the return address in the activation record). The approach can only detect known methods of attack, but the limited amount of instrumentation means the runtime overhead is low, making it suitable for use in deployed software.

MSE (Memory-Safety Enforcer): an approach to detect memory-safety violations, which are sometimes vulnerable to attacks, but are almost certainly programming errors. It can be used as a security tool on deployed software (checking only *writes*, with a low overhead), or as an efficient and effective debugging tool (checking both reads and writes, with a higher runtime overhead).

RTC (Runtime Type Checker): an approach to detect type-safety violations at runtime, including subtle errors that do not violate memory safety, and thus would not be detected by many existing approaches. The runtime overhead is quite high, making the approach impractical for use in deployed software, but suitable for debugging and testing.

Our tagged-memory approach tags each byte of memory with auxiliary information that is stored in a separate mirror space, which allows runtime checking to be applied without imposing restrictions on the flexibility of the C language. Instrumentation is performed at the source level, which makes the approach portable, and provides access to source-level information that enables the use of static analyses to eliminate unnecessary runtime checks. Efficient flow-insensitive analyses are shown to be effective at improving the runtime overheads significantly. Flow-sensitive analyses for identifying redundant checks and in-bounds dereferences further improve runtime performance, though they do not scale as well to larger programs.

Some of the weaknesses in our approach suggest possible directions for future research. Improved static analysis, in particular, in disambiguating structure fields in points-to analysis [Yon⁺99], could give rise to improvements in runtime performance. The *shadow-process* approach used for Guarding [Pat⁺97] could be adapted to our security tool, to execute the runtime checks in parallel with the regular program. Hardware approaches could also be used to improve performance. Hardware memory tagging has been proposed for encoding type information in Lisp machines [Moo03, Ste⁺87] and for data-access synchronization [Dal⁺92]. Specializing hardware to include tags for our approach would certainly improve performance, but would only be cost-effective if most or all programs being executed are instrumented with our tool. An alternative approach is to piggyback the tag bits onto existing hardware bits, such as the Error Correcting Code (ECC) bits of memory [Sch⁺94].

Another direction of future research is in improving the diagnostic capability of our error-detection tool. Adapting approaches to roll back the program state (including the state of the runtime tags) to an earlier point could facilitate the discovery of the source of a problem [Boo00].

The effectiveness of our tagged-memory approach at checking a variety of properties at runtime — from corruption of sensitive locations to memory and type safety — suggests as another direction for future research the extension of the tagging approach to check for

other properties. In runtime type-checking, tags could be augmented to check for higher-order type information [Sif⁺99]. In high-security applications, tags could be used for marking confidential data to prevent them from leaking to a non-confidential channel. For debugging or program understanding, tags could be used to trace the flow of values during program execution, or to help in computing the dynamic slice of a program [Kor⁺88]. Each new domain of application should give rise to interesting challenges in not only effecting the runtime checks, but in applying static analysis to eliminate checks that can be statically proven to be unnecessary.

APPENDIX

Summary of Benchmarks

The experimental results presented throughout this thesis use programs from four benchmark suites:

- Cyclone benchmarks (available from [Cyc]): programs used to evaluate the Cyclone tool [Jim⁺02]. The ones we tested are small but computationally intensive applications that make heavy use of arrays and pointers.
 - **aes**: Rijndael block-cipher encryption.
 - **cacm**: Adaptive arithmetic coding for data compression.
 - **cfrac**: Continued fraction algorithm.
 - **grobner**: Gröbner bases computation.
 - **matxmult**: Matrix multiplication.
 - **ppm**: Arithmetic encoding and decoding.
 - **tile**: Text document partitioning into “tiles”.
- Olden benchmarks (available from [Olden]): programs used to evaluate the Olden C compiler [Car⁺95]. These are relatively small programs that each perform a monolithic task, using a variety of dynamically allocated data structures.
 - **bh**: Barnes-Hut N-body force-computation algorithm; uses a heterogeneous octree.
 - **bisort**: Forward and backward sort of integers using 2 disjoint bitonic sequences that are merged to obtain the sorted result; uses a binary tree.

- **em3d**: Electromagnetic wave propagation in a 3D object; uses singly-linked lists.
- **health**: Columbian healthcare simulation; uses doubly-linked lists.
- **mst**: Minimum spanning tree of a graph; uses an array of singly-linked lists.
- **perimeter**: Perimeters of regions in images; uses a quad-tree.
- **power**: Power pricing system optimization problem solver; uses an N-way tree and singly-linked lists.
- **treeadd**: Recursive sum of values in a balanced B-tree.
- **tsp**: Traveling-salesman-problem solver using a partitioning algorithm and a closest point heuristic; uses a balanced binary tree.
- **Spec CPU95 [SPEC]**: includes all the C programs from the integer (CINT) suite.
 - **compress**: An in-memory version of the common UNIX utility.
 - **gcc**: Based on the GNU C compiler version 2.5.3.
 - **go**: An internationally ranked go-playing program.
 - **jpeg**: Image compression/decompression on in-memory images.
 - **li**: Xlisp interpreter.
 - **m88ksim**: A chip simulator for the Motorola 88100 microprocessor.
 - **perl**: An interpreter for the Perl language.
 - **vortex**: An object oriented database.
- **Spec CPU2000 [SPEC]**: includes select C programs from both the integer (CINT) and floating point (CFP) suites.
 - **ampp** (CFP): Computational chemistry.
 - **art** (CFP): Image recognition / neural networks.
 - **bzip2**: Compression.

- **crafty**: Game playing: chess.
- **equake** (CFP): Seismic wave propagation simulation.
- **gap**: Group theory, interpreter.
- **gzip**: Compression.
- **mcf**: Combinatorial optimization.
- **mesa** (CFP): 3-D graphics library.
- **parser**: Word processing.
- **twolf**: Place and route simulator.
- **vpr**: FPGA circuit placement and routing.

Tables A.1 and A.2 list the programs used in our experiments, along with their size (in lines of code), baseline compilation time (wallclock time, in seconds), and baseline execution times (wallclock time, in seconds). Inputs were selected to give reasonable running times for comparison: for the Cyclone and Olden benchmarks, the inputs we used are listed in Table A.1 (either command-line arguments, or input files supplied with the benchmarks). For the SPEC benchmarks, we used two different datasets, which we call the *slow* and *fast* datasets. The *slow* dataset is the **ref** set for Spec 95 and the **train** set for Spec 2000, and is used to evaluate the more efficient Memory-Safety Enforcer (MSE) and Sensitive Location Checker (SLC), in Chapters 3–7. The *fast* dataset is the **train** set for Spec 95 and the **test** set for Spec 2000, and is used to evaluate the slower Runtime Type Checker (RTC) in Chapters 9–10. In Figure A.2, columns (c) and (d) give the baseline execution times for the *slow* and *fast* datasets respectively.

The programs were compiled with `gcc` (version 3.3.2) and executed on a 1GHz Pentium III with 512MB RAM, running Linux (RedHat 9). For the MSE and SLC experiments, programs were compiled with `-O3` optimizations, while for the RTC experiments, optimizations were disabled (`-O0`) because they slowed down compilation time considerably, and we felt that the typical usage of the RTC as a debugging tool would be to compile programs without optimization.

| Program | LOC (a) | Compile Time (s) (b) | Exec Time (s) (c) | Input (d) |
|-----------|------------|----------------------------|-------------------------|--|
| Cyclone | | | | |
| aes | 1,822 | 1.67 | 0.83 | encode test2 4175764634412486014593803028771 eg03 2468 decode test1 sample2 |
| cacm | 340 | 0.36 | 0.67 | |
| cfrac | 4,218 | 3.76 | 1.69 | |
| grobner | 4,737 | 4.41 | 17.30 | |
| matxmult | 1,377 | 0.24 | 1.07 | |
| ppm | 1,421 | 1.45 | 0.65 | |
| tile | 4,880 | 1.57 | 0.33 | |
| Olden | | | | |
| bisort | 690 | 0.71 | 1.26 | 350000 0 |
| em3d | 538 | 0.69 | 2.33 | 2000 100 100 |
| health | 706 | 0.88 | 2.29 | 5 500 1 1 |
| mst | 610 | 0.73 | 2.58 | 2048 1 |
| perimeter | 472 | 0.66 | 0.37 | 11 1 |
| power | 867 | 0.97 | 2.46 | 1 1 |
| treeadd | 375 | 0.43 | 2.41 | 21 10 |
| tsp | 684 | 0.81 | 1.67 | 100000 1 |

Table A.1 Benchmark Information

| Program | LOC (a) | Compile Time (s) (b) | Exec Time (s) | |
|-----------|------------|----------------------------|--------------------|--------------------|
| | | | <i>slow</i> (c) | <i>fast</i> (d) |
| Spec 95 | | | (ref) | (train) |
| compress | 3,900 | 0.78 | 84.39 | 0.07 |
| gcc | 205,106 | 125.41 | 47.27 | 2.50 |
| go | 29,629 | 20.67 | 102.40 | 0.85 |
| jpeg | 31,215 | 20.33 | 71.96 | 2.10 |
| li | 7,630 | 8.13 | 50.59 | 0.27 |
| m88ksim | 19,227 | 19.48 | 62.67 | 0.18 |
| perl | 26,872 | 30.63 | 42.86 | 1.18 |
| vortex | 67,219 | 53.42 | 82.39 | 3.27 |
| Spec 2000 | | | (train) | (test) |
| ammp | 13,483 | 13.88 | 140.75 | 27.29 |
| art | 1,270 | 1.51 | 96.81 | 30.13 |
| bzip2 | 4,650 | 3.79 | 78.54 | 20.52 |
| crafty | 20,545 | 23.82 | 32.45 | 6.91 |
| equake | 1,513 | 1.57 | 107.73 | 3.50 |
| gap | 71,363 | 46.21 | 10.05 | 2.01 |
| gzip | 8,605 | 4.61 | 40.92 | 3.65 |
| mcf | 2,412 | 2.62 | 60.37 | 0.58 |
| mesa | 58,724 | 52.27 | 118.50 | 4.17 |
| parser | 11,391 | 13.73 | 15.42 | 7.80 |
| twolf | 20,461 | 29.26 | 26.26 | 0.58 |
| vpr | 17,730 | 13.85 | 21.50 | 1.53 |

Table A.2 Benchmark Information

LIST OF REFERENCES

- [And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
- [Ash⁺02] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.
- [Asu92] Jonathan Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1992.
- [Aus⁺94] Todd Austin, Scott Breach, and Gurinder Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–201, Orlando, FL, June 1994.
- [Bal⁺01] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *13th Conference on Computer Aided Verification*, pages 260–264, Paris, France, July 2001.
- [Bar⁺00] Arash Baratloo, Timothy Tsai, and Navjot Singh. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, San Diego, CA, June 2000.
- [Bod⁺00] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, Vancouver, BC, June 2000.
- [Boo00] Bob Boothe. Efficient algorithms for bidirectional debugging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 299–310, Vancouver, BC, June 2000.
- [Bur⁺03] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 90–105, Warsaw, Poland, April 2003. Springer.

- [Bus⁺00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, June 2000.
- [Car⁺91] Robert Cartwright and Mike Fagan. Soft typing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, ON, June 1991.
- [Car⁺95] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Amsterdam, July 1995.
- [Car97] Luca Cardelli. Type systems. In Allen B. Tucker Jr., editor, *Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.
- [Cha⁺97] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. Types behind the mirror: A proposal for partial ML type reconstruction at runtime. In *Types In Compilation workshop*, Amsterdam, June 1997.
- [Che⁺04] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2004.
- [Chi⁺01] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing Systems*, Phoenix, AZ, April 2001.
- [Ckit] Ckit. <http://www.smlnj.org/doc/ckit/>.
- [Con⁺03] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, CA, June 2003.
- [Cou⁺76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *ACM Symposium on Principles of Programming Languages*, pages 106–130, April 1976.
- [Cow⁺98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, San Antonio, TX, January 1998.
- [Cow⁺00] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, volume 2, pages 119–129, Hilton Head, SC, January 2000.

- [Cyc] Cyclone. <http://www.research.att.com/projects/cyclone/>.
- [Dal⁺92] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor — a multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, 1992.
- [Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, June 2000.
- [DeL⁺01] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, June 2001.
- [Det⁺98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report SRC-159, Compaq SRC, 1998.
- [Dhu⁺03] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 69–80, San Diego, CA, June 2003.
- [Dor⁺01] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, page 194, Paris, France, July 2001. Springer.
- [Dvo00] Dvorak. LBL traceroute exploit, May 2000. Posting to BugTraq.
- [Dyn] DynamoRIO. <http://cag.lcs.mit.edu/dynamorio/>.
- [Ema⁺94] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994.
- [Eva96] David Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, May 1996.
- [Gro⁺02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Hanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

- [Gup93] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, March–December 1993.
- [Har77] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.
- [Har⁺03] Matthew Harren and George C. Necula. Lightweight wrappers for interfacing with binary code in CCured. In *Proceedings of the International Symposium on Software Security*, Tokyo, Japan, November 2003.
- [Has⁺92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, pages 125–138, San Francisco, CA, January 1992.
- [Hau⁺03] Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2003.
- [Hen92] Fritz Henglein. Global tagging optimization by type inference. In *LISP and Functional Programming*, pages 205–215, San Francisco, CA, June 1992.
- [Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.
- [Jim⁺02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.
- [Jon⁺97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, pages 13–26, Linköping, Sweden, May 1997.
- [Kau⁺88] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C: An interpreter-based programming environment for the C language. In *USENIX Summer Conference*, pages 161–171, San Francisco, CA, June 1988.
- [Kee⁺02a] Diana Keen, Frederic T. Chong, Premkumar Devanbu, and Matthew Farrens. Extending commodity microprocessors for software safety: An experiment. Technical Report CSE-2002-3, UC Davis, 2002.
- [Kee⁺02b] Diana Keen, Foo Lim, Frederic T. Chong, Premkumar Devanbu, Matthew Farrens, Paul Sultana, Cathy Zhuang, , and Ravishankar Rao. Hardware support for pointer safety in commodity microprocessors. Technical Report CSE-2002-1, UC Davis, 2002.

- [Ken83] Samuel C. Kendall. Bcc: Runtime checking for C programs. In *Proceedings of the USENIX Summer Conference*, pages 5–16, Toronto, Canada, July 1983.
- [Kir⁺02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, San Francisco, CA, August 2002.
- [Kol⁺95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–278, La Jolla, CA, June 1995.
- [Kor⁺88] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.
- [Kow⁺02] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, October 2002.
- [Lan⁺91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.
- [Lan⁺92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, CA, June 1992.
- [Lar⁺01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, Washington, DC, August 2001.
- [Lar⁺03] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *12th USENIX Security Symposium*, Washington, DC, August 2003.
- [Liv⁺03] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *European Software Engineering Conference (ESEC) and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 317–326, Helsinki, Finland, September 2003.
- [Luj⁺02] Mikel Luján, John R. Gurd, T. L. Freeman, and José Miguel. Elimination of Java array bounds checks in the presence of indirection. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 76–85, Seattle, WA, November 2002.

- [Mar⁺82] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 17(6)*, pages 114–119, Boston, MA, June 1982.
- [McG97] Greg McGary. Technical specifications for bounded pointers in GCC, 1997. <http://gcc.gnu.org/projects/bp/main.html>.
- [Mil⁺95] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison, 1995.
- [Moo⁺02] David Moore, Colleen Shannon, and Jeffery Brown. Code-Red: A case study on the spread and victims of an Internet worm. In *Proceedings of the second ACM SIGCOMM Workshop on Internet Measurement*, pages 273–284, Marseille, France, November 2002.
- [Moo03] David A. Moon. Architecture of the Symbolics 3600. In *International Symposium on Computer Architecture*, pages 76–83, Boston, MA, June 2003.
- [Nec⁺02] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002.
- [Nec04] George C. Necula, August 2004. Personal Communication (e-mail).
- [Net⁺03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Third Workshop on Runtime Verification (RV'03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*, Boulder, CO, 2003.
- [Oiw⁺02] Yutaka Oiwa, Tatsuro Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C compiler: An approach to making C programs secure (progress report). In *Proceedings of the International Symposium on Software Security*, volume 2609 of *Lecture Notes in Computer Science*, pages 133–153, Tokyo, Japan, November 2002. Springer.
- [Olden] Olden. <http://www.cs.princeton.edu/~mcc/olden.html>.
- [Özd⁺02] Hilmi Özdoğanoglu, Carla E. Brodley, T. N. Vijaykumar, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A hardware solution to prevent security attacks on the function return address. Technical Report TR-ECE 03-13, School of Electrical and Computer Engineering, Purdue University, December 2002.
- [Pac] Packet Storm. <http://packetstormsecurity.org/>.

- [Par⁺92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127, San Francisco, CA, June 1992.
- [Pat⁺97] 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.
- [Ros86] Graham Ross. Integral-C — a practical environment for C programming. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, Palo Alto, CA, December 1986.
- [Rug⁺88] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *ACM Symposium on Principles of Programming Languages*, pages 285–293, San Diego, CA, January 1988.
- [Ruw⁺04] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium (NDSS)*, pages 159–169, San Diego, CA, February 2004.
- [Sch⁺94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, San Jose, CA, 1994.
- [scu01] scut. Exploiting format string vulnerabilities, 2001. TESO Security Group.
- [Sha⁺97] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 115–134, Paris, France, September 1997. Springer.
- [Sif⁺99] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with type casts in C. In *European Software Engineering Conference (ESEC) and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 180–198, Toulouse, France, September 1999.
- [Smi97] Nathan P. Smith. Stack smashing vulnerabilities in the UNIX operating system. Technical report, Computer Science Department, Southern Connecticut State University, 1997.
- [Sol] Solar Designer. Openwall project. <http://www.openwall.com/>.
- [sor02] sorbo. Exploit for traceroute 1.4a5, October 2002. Available from PacketStorm, <http://packetstormsecurity.org/>.

- [SPEC] Standard Performance Evaluation Corporation. <http://www.spec.org/>.
- [Sta00] Stack Shield, January 2000. Technical info file, Version 0.7.
- [Sta⁺02] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB: the GNU Source-Level Debugger*. Free Software Foundation, 9th edition, January 2002.
- [Ste⁺87] Peter Steenkiste and John Hennessy. Tags and type checking in LISP: Hardware and software approaches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59, Palo Alto, CA, October 1987.
- [Ste92] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software—Practice and Experience*, 22(4):305–316, April 1992.
- [Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 1996.
- [Sue⁺03] Kohei Suenaga, Yutaka Oiwa, Eijiro Sumii, and Akinori Yonezawa. The interface definition language for Fail-Safe C. In *Proceedings of the International Symposium on Software Security*, Tokyo, Japan, November 2003.
- [Wag⁺00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed Systems Security (NDSS)*, pages 3–17, San Diego, CA, February 2000.
- [Wel78] Jim Welsh. Economic range checks in Pascal. *Software—Practice and Experience*, 8:85–97, 1978.
- [Wil⁺95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995.
- [Wil⁺03] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Network and Distributed System Security Symposium (NDSS)*, pages 149–162, San Diego, CA, February 2003.
- [Yon⁺99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, Atlanta, GA, May 1999.
- [Yon⁺04] Suan Hsi Yong and Susan Horwitz. Pointer-range analysis. In *Static Analysis Symposium*, Verona, Italy, August 2004. (To appear).

Index

- alias-locs*, 50, 51
- array-descriptor domain, 59
- assignment graph, 126
- assignment operator, 107
- `atexit` table, 14
- buffer overrun, 4, 15
- cast
 - conversion, 110
 - copy, 110
- CCured, 27, 84, 109, 149
- $Checked_{in}^n$, $Checked_{out}^n$, 46, 142
- checked-derefs*
 - EPT, 40
 - MSE, 31
 - naive, 32
 - range analysis, 76
 - redundant checks, 47
 - SLC, 94
 - write-only, 33
- chunk, malloc, 18
- classification, MSE, 31
- continuation bit, 112
- conversion cast, 110
- copy cast, 110
- CopyTag, 115
- coverage, 33
 - tracked, 80
- Cyclone, 84, 149
- data bits, 112
- demand-zero paging, 28, 95
- dereference, 24
 - checked, 31
 - definitely safe, 34
 - potentially unsafe, 34
 - unchecked, 31
- derefs*, 30
- descriptor-offset domain, 60
- DO (Descriptor-Offset), 60
- `.dtors` table, 14
- Env*, 61
- EPT (Extended Points-To) analysis, 38
- Eval*, 62
- `exec`, 14, 94
- fat pointer, 26
- format string vulnerability, 16
- function pointers, 14

- Gen/Kill*, 47, 142
- global offset table, 14, 19
- GOT (global offset table), 14, 19
- Hobbes, 109, 151
- init*, 112
- instrumentation, MSE, 31
- integer interval domain, 56
- INTEGRAL, 111
- intended target, 23
- intersect*, 69
- invalid access, 23
 - invalid read, 23
 - invalid write, 23
- Kill*, 47, 142
 - Kill_{lval}*, *Kill_{type}*, 142
- LO* (Location-Offset), 57
- location, 123
- location-offset domain, 57
- locs*, 30, 57, 123
- locs_s*, 74
- locs_u*, 60
- longjmp buffers, 14, 94
- lvalue*, 47
- lvalue* expression, 49, 107
- lvalue-affecting-locs*, 50, 51
- malloc chunk, 18
- MALLOC_{*i*}, 30
- may-be-uninitialized analysis
 - flow-insensitive, 41
 - flow-sensitive, 137
- may-points-to analysis, 36
- MayFree*, 47
- MayMod*, 47, 138
- MBU (may-be-uninitialized) analysis, 139
- MBU-lite, 140
- mem-unsafe expression, 123
- mirror
 - MSE, 28
 - RTC, 112
- MSE (Memory-Safety Enforcer), 23
- narrowing/widening, 71
- open array, 26, 28
- open-max*, 70
- open-min*, 70
- outer*, 75
- outermost object, 25, 31
- pointer*, 112
- points-to analysis, 36
- popen, 14
- possible-type*, 128, 129
- produce*, 107
- pt-set*, 36, 50, 51
- Purify, 101, 151

REAL, 111
 redundant checks analysis, 46, 141
required-type, 128, 129
 return address, 14
 RTC (Runtime Type Checker), 101
 runtime type, 108, 111
rvalue, 47
rvalue-affecting-locs, 50, 51
 safe expression, 123
 sensitive location, 13, 82, 90
SetTag, 115
 shadow process, 149, 156
 shellcode, 13
 SLC (Sensitive Location Checker), 90
 spatial access error, 23
 stack smashing, 4, 14
 STRLIT_{*i*}, 30
system, 14
T#, 127
 lattice, 128
 tag
 bit value, 28, 95
 runtime type, 112
 valid/invalid, 28, 90
 temporal access error, 23
traceroute vulnerability, 18
 tracked coverage, 80
 tracked location
 MSE, 31
 RTC, 123
 SLC, 94
tracked-locs
 MSE, 31
 naive, 32
 points-to analysis, 37
 SLC, 94
 type class, 112
 type-unsafe expression, 123
unalloc, 109, 112
uninit, 108, 112
Uninit_{in}ⁿ, *Uninit_{out}ⁿ*, 137
 unsafe location, 123
 untracked location
 MSE, 31
 RTC, 123
use, 107
 VALUE_{*τ*}, 127
Var, 61
VerifyDeref, 115
VerifyType_assign, 115
VerifyType_use, 115
 widening/narrowing, 71
write-derefs, 33