Accelerating Dynamic Detection of Memory Errors for C Programs via Static Analysis

by

Ding Ye

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN THE SCHOOL OF COMPUTER SCIENCE AND ENGINEERING THE UNIVERSITY OF NEW SOUTH WALES SYDNEY, AUSTRALIA

Tuesday 10th February, 2015

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

© Ding Ye 2015
Memory errors in C programs are the root causes of many defects and vulnerabilities in software engineering. Among the available error detection techniques, dynamic analysis is widely used in industries due to its high precision. Unfortunately, existing approaches suffer from considerable runtime overheads, owing to unguided and overly conservative instrumentation. With the massive growth of software nowadays, such inefficiency prevents testing with comprehensive program inputs, leaving some input-specific memory errors undetected.

This thesis presents novel techniques to address the efficiency problem by eliminating some unnecessary instrumentation guided by static analysis. Targeting two major types of memory errors, the research has developed two tools, Usher and WPBound, both implemented in the LLVM compiler infrastructure, to accelerate the dynamic detection.

To facilitate efficient detection of undefined value uses, Usher infers the definedness of values using a value-flow graph that captures def-use information for both top-level and address-taken variables inter-procedurally, and removes unnecessary instrumentation by solving a graph reachability problem. Usher works well with any pointer analysis (done a priori) and enables advanced instrumentation-reducing optimizations.

For efficient detection of spatial errors (e.g., buffer overflows), WPBound enhances the performance by reducing unnecessary bounds checks. The basic idea is to guard a bounds check at a memory access inside a loop, where the guard is computed outside the loop based on the notion of weakest precondition. The falsehood of the guard implies the absence of out-of-bounds errors at the dereference, thereby avoiding the corresponding bounds check inside the loop.

For each tool, this thesis presents the methodology and evaluates the implementation with a set of C benchmarks. Their effectiveness is demonstrated with significant speedups over the state-of-the-art tools.
ORIGINALITY STATEMENT

‘I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.’

Signed

Date 10/Feb/2015
COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.
I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).
I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed

Date 10/Feb/2015

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed

Date 10/Feb/2015
Abstract

Memory errors in C programs are the root causes of many defects and vulnerabilities in software engineering. Among the available error detection techniques, dynamic analysis is widely used in industries due to its high precision. Unfortunately, existing approaches suffer from considerable runtime overheads, owing to unguided and overly conservative instrumentation. With the massive growth of software nowadays, such inefficiency prevents testing with comprehensive program inputs, leaving some input-specific memory errors undetected.

This thesis presents novel techniques to address the efficiency problem by eliminating some unnecessary instrumentation guided by static analysis. Targeting two major types of memory errors, the research has developed two tools, Usher and WPBound, both implemented in the LLVM compiler infrastructure, to accelerate the dynamic detection.

To facilitate efficient detection of undefined value uses, Usher infers the definedness of values using a value-flow graph that captures def-use information for both top-level and address-taken variables interprocedurally, and removes unnecessary instrumentation by solving a graph reachability problem. Usher works well with any pointer analysis (done a priori) and enables advanced instrumentation-reducing optimizations.

For efficient detection of spatial errors (e.g., buffer overflows), WPBound en-
hances the performance by reducing unnecessary bounds checks. The basic idea is to guard a bounds check at a memory access inside a loop, where the guard is computed outside the loop based on the notion of weakest precondition. The falsehood of the guard implies the absence of out-of-bounds errors at the dereference, thereby avoiding the corresponding bounds check inside the loop.

For each tool, this thesis presents the methodology and evaluates the implementation with a set of C benchmarks. Their effectiveness is demonstrated with significant speedups over the state-of-the-art tools.
Publications


• Ding Ye, Yu Su, Yulei Sui and Jingling Xue. WPBound: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. *IEEE International Symposium on Software Reliability Engineering (ISSRE ’14).*

• Yu Su, Ding Ye and Jingling Xue. Parallel Pointer Analysis with CFL-Reachability. *IEEE International Conference on Parallel Processing (ICPP ’14).*

• Yulei Sui, Ding Ye and Jingling Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering (TSE ’14).*

• Ding Ye, Yulei Sui and Jingling Xue. Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis. *IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’14).*

• Yu Su, Ding Ye and Jingling Xue. Accelerating Inclusion-based Pointer Analysis on Heterogeneous CPU-GPU Systems. *IEEE International Conference on High Performance Computing (HiPC ’13).*
• Yulei Sui, Ding Ye and Jingling Xue. Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis. *International Symposium on Software Testing and Analysis (ISSTA ’12).*

• Peng Di, Ding Ye, Yu Su, Yulei Sui and Jingling Xue. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. *IEEE International Conference on Parallel Processing (ICPP ’12).*
Acknowledgements

I would like to express my sincere gratitude to everyone who has supported me throughout my PhD study. This thesis is a direct result of their constructive guidance, vigorous assistance, and constant encouragement.

My supervisor, Prof. Jingling Xue, has made an invaluable impact on me. His trust, patience, enthusiasm, and broad knowledge have helped me develop further in the field of computer science. He leads the CORG group, inspiring us to believe that rewards will eventually come if we stay on the right track, and thus creates a positive research atmosphere. In addition, he always spends a great deal of time and energy with me, discussing research ideas, and refining and polishing paper writing. Without his physical, mental and spiritual help, I would not have been able to enjoy such an excellent level of supervision from the very start.

I would like to thank my group mates Yulei Sui and Peng Di for their dedicated collaboration on my work. Yulei is a kind person who is always willing to share. I have learned a lot about program analysis theory, coding skills, and a passion for being a programming expert from him. Peng has also inspired me greatly with his devoted focus on his research.

I am also very much thankful to all of the other members of the CORG group, past and present — Yi Lu, Xinwei Xie, Lei Shang, Qing Wan, Sen Ye, Yue Li, Hao Zhou, Tian Tan, Xiaokang Fan, Hua Yan, and Feng Zhang. I have had an
amazing and pleasant research experience whilst working with them. Also many thanks to Manuel Chakravarty, Michael Thielscher, Eric Martin, Hui Wu, and June Andronick for being my annual progress review panel members.

I acknowledge the funds I received for my study, living allowance and travel — UIPA of UNSW, PRSS of UNSW, and ACM travel grants. The research for this thesis has also been funded by Australian Research Grants (DP110104628 and DP130101970), and a generous gift by Oracle Labs.

Finally, I give my special gratitude to my wife, Yu Su, for her unconditional love. I would also like to thank my parents for providing me with such a good education and so much more.
# Contents

Abstract i

Acknowledgements v

List of Figures xii

List of Tables xiii

List of Algorithms xiv

1 Introduction 1

1.1 Challenges 3

1.2 Our Approaches 4

1.3 Contributions 5

1.4 Thesis Organization 6

2 Background 8

2.1 Memory Errors in C Programs 8

2.1.1 Memory Error Classification 9

2.1.2 Impacts of Memory Errors 10

2.1.3 Alternative languages to C 12

2.2 Detecting Memory Errors at Runtime 13
3.4.4 Value-Flow Analysis .................................... 48
3.4.5 Instrumentation Overhead .................................. 52
3.4.6 Effect of Compiler Optimizations on Reducing Instrumentation Overhead .................................. 55
3.5 Related Work .................................................. 56
  3.5.1 Detecting Uses of Undefined Values ...................... 56
  3.5.2 Combining Static and Dynamic Analysis ................... 57
  3.5.3 Value-Flow Analysis .................................. 58
  3.5.4 Pointer Analysis .................................. 58
3.6 Chapter Summary ............................................. 59

4 Accelerating Detection of Spatial Errors .................. 60
  4.1 Overview ................................................. 60
  4.2 Background ............................................. 64
  4.3 The WPBOUND Framework .................................. 66
    4.3.1 A Motivating Example ................................ 68
    4.3.2 The LLVM IR .................................. 72
    4.3.3 Value Range Analysis ................................ 74
    4.3.4 WP-based Instrumentation .......................... 77
  4.4 Evaluation .............................................. 83
    4.4.1 Implementation Considerations ....................... 84
    4.4.2 Experimental Setup .................................. 85
    4.4.3 Methodology .................................. 85
    4.4.4 Instrumentation Results ................................ 86
    4.4.5 Performance Results ................................ 89
  4.5 Related Work ............................................. 92
    4.5.1 Guard Zone-based Spatial Safety ....................... 93
List of Figures

2.1 Vulnerable code with a possible undefined value. ................. 10
2.2 Vulnerable code with spatial safety threats. ...................... 11
2.3 LLVM compilation toolchain. .................................. 20

3.1 The TinyC source language. .................................. 29
3.2 The TinyC representation vs. LLVM-IR. ......................... 30
3.3 The Usher value-flow analysis framework. ....................... 31
3.4 The TinyC language in SSA form. .......................... 32
3.5 A TinyC program and its SSA form. .......................... 33
3.6 An example of a semi-strong update. ......................... 35
3.7 Instrumentation rules for $\top$-nodes. ....................... 38
3.8 Instrumentation rules for $\perp$-nodes. ....................... 39
3.9 Instrumentation rules for virtual nodes. ...................... 40
3.10 An example of value-flow simplification. .................... 44
3.11 An example of redundant check elimination. ................ 44
3.12 Execution time slowdowns. ................................ 53
3.13 Static numbers of shadow operations. ....................... 54

4.1 Reported buffer overflow vulnerabilities in the past decade. .... 61
4.2 Pointer-based instrumentation with disjoint metadata. .......... 65
4.3 Overview of the WPBOUND framework. ....................... 67
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>A motivating example.</td>
<td>70</td>
</tr>
<tr>
<td>4.5</td>
<td>The LLVM-IR (in pseudocode) for a C program.</td>
<td>73</td>
</tr>
<tr>
<td>4.6</td>
<td>Range analysis rules.</td>
<td>75</td>
</tr>
<tr>
<td>4.7</td>
<td>Compilation workflow.</td>
<td>86</td>
</tr>
<tr>
<td>4.8</td>
<td>Bitcode file sizes after instrumentation.</td>
<td>89</td>
</tr>
<tr>
<td>4.9</td>
<td>Execution time.</td>
<td>90</td>
</tr>
<tr>
<td>4.10</td>
<td>Percentage of dynamic number of checks.</td>
<td>92</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Performance of USHER’s value-flow analysis. .................. 48
3.2 Variable statistics. ........................................ 49
3.3 Updates performed on stores. .............................. 50
3.4 Value-flow graph statistics. ................................. 51

4.1 Benchmark statistics. ........................................ 87
4.2 Static instrumentation results. ............................. 88
List of Algorithms

1. Redundant Check Elimination ........................................ 45
2. Loop-Directed WP Abstraction ...................................... 79
3. WP Consolidation ...................................................... 81
4. WP-Driven Loop Unswitching ....................................... 82
Chapter 1

Introduction

The C programming language allows programmers to take explicit low-level control of memory access and management. Such features aim to exploit the full performance of the underlying hardware. Beyond its original purpose of building the UNIX operating system, C has become a desirable language for performance-critical software development. A large body of software projects is written in C (and/or its variants), together with their ecosystems of libraries and utilities.

While providing the low-level performance-oriented features, the C programming language suffers from a major drawback of lacking memory safety, since it does not ensure the correctness and safety of the low-level memory controls. The resulting memory errors can be the root causes of incorrect computations, system crashes, and security vulnerabilities \[61, 76, 77\]. For example, for efficient memory allocation, stack variables are declared without initialization. They make room for code injection attacks before being explicitly defined by programmers. In addition, for an efficient memory access, a pointer dereference is performed without checking whether the accessed memory location is within the legal range pointed by the pointer. It may access another (adjacent) variable or cause a silent memory
corruption when there is an out-of-bounds access.

Despite the memory errors of all kinds, C and its variants are still widely used nowadays in many domains due to their primary focus on performance. C and C++ are commonly used in a wide range of software development, including operating systems, compilers, virtual machines, database management systems, web browsers, game engines, and some desktop applications and embedded software; Objective-C is one of the dominant languages for writing mobile apps [43]; and CUDA is becoming increasingly popular in parallel programming [69]. It is crucial to detect memory errors efficiently and effectively, in order to eliminate the entire class of bugs caused by memory safety violations.

To tackle memory errors, program analysis techniques have been proposed in both static and dynamic terms. On one hand, static analysis tries to approximate the runtime behavior of a program. It is difficult to achieve both accuracy and scalability due to its approximation nature, and thus the analyzers usually report false alarms [3, 8, 9, 18, 19, 22, 26, 29, 31, 40, 45, 47, 55, 73, 74, 89, 100]. On the other hand, dynamic analysis uses program instrumentation to monitor the exact program behavior on the fly. It appears to be a more practical and promising method [66, 76, 110], since the errors observed during the program execution can be precisely captured. However, significant performance overheads are introduced due to the execution of extra instructions, limiting the code coverage of testing in practice.

This thesis attempts to boost the performance of dynamic error detection via static program analysis. We propose two different approaches, which are able to deal with a broad class of memory errors. Both approaches explore novel techniques to achieve efficient instrumentation for detecting the focused errors. Implemented in the LLVM compiler infrastructure and demonstrated on a suite of SPEC 2000
and 2006 benchmarks, our approaches can significantly speed up existing dynamic error detection tools.

1.1 Challenges

This thesis aims at accelerating runtime memory error detection via static analysis. It involves a range of program analysis techniques in both static and dynamic terms, confronted with the following challenges:

- **Diversity of Memory Errors.** Memory errors are among the most difficult programming bugs to track down due to the delayed observable symptoms. Different types of memory errors usually have distinct characteristics, and they require different techniques for detection [7, 66]. For example, def-use analysis is suitable for detecting undefined value uses, while bounds range information is more useful for spatial error detection.

- **Efficiency of Instrumentation.** Detecting memory errors at runtime via instrumentation is usually costly and brings significant overheads. A program is instrumented with shadow code, which monitors memory safety by maintaining and propagating metadata during program execution and performing checks for some potentially safety-violating operations. Such shadow code makes the program become inefficient. For example, to detect uses of undefined values, a state-of-the-art tool MSan typically incurs a 3X slowdown [30]. It is challenging but crucial to introduce efficient program instrumentation as it helps improve code coverage of a memory error detection tool.

- **Efficiency of Static Analysis.** A great deal of the shadow code in existing memory error detection tools can be proven unnecessary via static analysis,
and they can be thus eliminated to improve the instrumentation efficiency. As an approximation of program runtime behaviors, static analysis is supposed to be efficient at compile-time for large-scale programs, and meanwhile informative enough to guide dynamic analysis effectively. It is challenging yet promising to make a trade-off between its precision and scalability [26, 47, 89].

1.2 Our Approaches

To address the challenges listed above, this thesis proposes two compiler optimizations to speed up the runtime detection of two typical types of memory errors, respectively. They are both implemented in the LLVM compiler infrastructure, providing efficient source-level instrumentation. However, they use orthogonal techniques to tackle different problems with distinct features.

- **Accelerating Detection of Undefined Value Uses.** We perform an interprocedural static value-flow analysis to guide the instrumentation for efficient detection of undefined value uses. The value-flow analysis, coupled with a points-to analysis, utilizes memory SSA to compute and optimize def-use chains, which are used to infer the definedness of all values in a program by resolving a graph reachability problem. By this means, the definedness of a large number of the program values can be statically proven, and their corresponding shadow code can be eliminated for efficient instrumentation. To achieve sufficient precision while being scalable, such analysis is field-, flow- and context-sensitive wherever appropriate and supports two flavors of strong updates.

- **Accelerating Detection of Spatial Errors.** We develop a method for optimizing pointer bounds checks based upon static program analysis, which is
used to reduce the amount of instrumentation required for detecting spatial errors. The central concept of this idea is to statically infer a loop invariant that approximates a weakest precondition of a spatial error occurrence inside the loop. Such weakest precondition is computed based on a value range analysis, leveraging LLVM’s Scalar Evolution analysis. By checking this precondition outside of a loop, we can ensure the loop will execute error free. As a result, runtime checks for every loop iteration can be avoided, and the instrumentation becomes more efficient.

Although this thesis focuses on the detection of undefined value uses and spatial errors (e.g., buffer overflows), these two types of chosen errors cover the two different sets of characteristics of all memory errors. Our approaches are expected to work for other types of memory errors as well, with appropriate extensions that will be discussed in Chapter 5.

### 1.3 Contributions

We have developed two compiler optimization tools Usher and WPBOUND to tackle undefined value uses and spatial errors, respectively, thereby making the following contributions.

- **The Usher Tool**

  Usher is a new static value-flow analysis approach for detecting the uses of undefined values in C programs. It statically infers the definedness of values using a value-flow graph (VFG) that captures def-use chains for all variables interprocedurally and removes unnecessary instrumentation by solving a graph reachability problem. The value-flow analysis is sound (by missing no bugs statically) as long as the underlying pointer analysis is. **Usher**
represents the first such whole-program analysis for handling top-level and address-taken variables to guide dynamic instrumentation for C programs. Usher’s VFG representation allows advanced instrumentation-reducing optimizations to be developed (with two demonstrated in this thesis). In addition, its precision can be improved orthogonally by leveraging existing and future advances on pointer analysis.

Implemented in LLVM, Usher can reduce the slowdown of MSan from 212% – 302% to 123% – 140% for all the 15 SPEC2000 C programs under a number of configurations tested.

• The WPBOUND Tool

WPBOUND is a weakest precondition-based source-level instrumentation tool for efficiently enforcing spatial safety for C programs. The weakest preconditions of spatial errors are approximated in a conservative manner, and are then used to accelerate runtime spatial safety enforcement by reducing some unnecessary bounds checking.

Implemented in LLVM, WPBOUND’s optimization leverages LLVM’s analysis passes to compute the weakest preconditions.

As evaluated on a set of 12 C programs, WPBOUND reduces SoftBound’s average runtime overhead from 71% to 45% (by a reduction of 37%), with small code size increases.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

In Chapter 2, we initially present a detailed background about memory errors
in C programs. Then we provide a comprehensive survey of existing work related to the problems. Next, we briefly explain how program instrumentation works for error detection. Lastly, we introduce some relevant LLVM knowledge, based on which our experiments are carried out.

In Chapters 3 and 4, we separately present two novel techniques to tackle undefined value uses and spatial errors. Chapter 3 shows how we perform static value-flow analysis to infer the definedness of every value in a program, and how some unnecessary instrumentation can be eliminated. Chapter 4, with a motivating example, illustrates how we conservatively approximate weakest preconditions of spatial errors at compile-time, and how these approximated weakest preconditions can be used to reduce instrumentation overheads. The techniques proposed in these two chapters were previously published in [105] and [104], respectively.

In Chapter 5, we conclude the thesis and discuss possible extensions and future research.
Chapter 2

Background

This chapter presents some background knowledge that is closely related to our study. We initially introduce in details the memory errors in C programs in Section 2.1. We then describe the error detection techniques in Section 2.2, and, especially, how the instrumentation is performed for the runtime detection in Section 2.3. As we use LLVM as our compiler for implementation, we also provide background of LLVM and its intermediate representation in Section 2.4.

2.1 Memory Errors in C Programs

The C programming language was originally designed for writing operating systems. The spirit behind is to make it a simple and efficient language that can be easily mapped to typical machine instructions. Unlike some application-level programming languages (e.g., Java), C, together with its OO incarnation C++, is generally used for system-level or performance-critical code programming. It allows programmers to take the low-level control of memory layout and access, with features including arbitrary type casts, array-pointer conflation, and manual memory management. In such a design, more focus is given to the code efficiency than
to the safety and security aspects. As a result, memory errors are common in C programs.

### 2.1.1 Memory Error Classification

To the best of our knowledge, there is not any formal definition of memory errors. Typically, memory-related programming errors for C/C++ can be broadly classified in the following categories:

- **Undefined Value Use.** It happens if a program uses some value from a variable that has been allocated but not initialized. When stack variables or heap memory chunks returned by `malloc()`, `new` and `new[]` functions are allocated, they immediately carry garbage values, which are the roots of the undefinedness.

- **Spatial Error.** It occurs when a pointer is dereferenced where the memory location outside of the object it points to is accessed. Basically, there are three scenarios for a spatial safety violation:
  - Dereferencing an uninitialized pointer or a null pointer.
  - Dereferencing non-pointer data. Arbitrarily casting integers to pointers is a common example.
  - Dereferencing a valid pointer with an out-of-bounds address. Such pointers are generally obtained from invalid pointer arithmetic for buffer-like variables, and such errors are also referred to as buffer overflows.

- **Temporal Error.** It takes place by using a dangling pointer whose pointee has already been de-allocated. Typical temporal safety violations are use-after-free (i.e., dereferencing dangling pointers) and double-free (i.e., passing dangling pointers to function `free()`) errors.
• **Memory leak.** It happens when dynamically allocated memory that is no longer needed is not released.

### 2.1.2 Impacts of Memory Errors

Memory errors are usually the underlying root causes of program performance issues, computational incorrectness, system crashes, and security vulnerabilities.

Memory leaks can use up memory resources of a system when the application runs, bringing negative impact on performance and system reliability. It is especially harmful for long-running server applications. Undefined value uses, as well as spatial and temporal errors can often crash a program immediately, or lead to unexpected computational results.

```c
void foo() {
    struct my_state s;
    s.flag = 1;
    if (COND) s.body = ...;
    ...
    if (s.body == ...) {
        // do something
    } else {
        // do something else
    }
}
```

Figure 2.1: Vulnerable code with a possible undefined value.

To make things worse, severe security problems can occur when undefined values or buffer overflows are exploited. Figure 2.1 shows a vulnerable code snippet with an uninitialized variable. For function `foo()`, the stack variable `s` at line 2 is allocated without initialization. Later on, the `flag` field is defined at line 4, while
Chapter 2. Background

the body field is conditionally defined at line 5. When the condition COND is false, the body field stays undefined. An attack may inject some intended value into s.body before function foo() is invoked, and thus takes control of the program execution on either the branch at line 10 or 12.

```
1 | int bar() {
2 |     char buffer[16];
3 |     int pass = 0;
4 |
5 |     ... |
6 |     gets(buffer);
7 |     if (!strcmp(buffer, "correct_pwd")) {
8 |         pass = 1;
9 |     }
10 |
11 |     if (pass) {
12 |         printf("Access Granted!");
13 |     }
14 |
15 |     return pass;
16 | }
```

Figure 2.2: Vulnerable code with spatial safety threats.

Another example is described in Figure 2.2, with opportunities of buffer overflow attacks. Function bar() verifies if the user input password equals the correct one. It has two stack variables, an array buffer and an integer pass with its initialized value 0. It takes the user input string into buffer at line 7. If this input equals the correct password, the value of pass is updated to 1 (lines 8 – 10). Finally, if the value of pass is found to be non-zero, the access is granted (lines 12 – 14). Based on this code snippet, an attack would input some string of 17 chars to trigger a buffer overflow error at line 7. Such an attack not only modifies the variable buffer, but also changes the value of its adjacent variable pass to be non-zero.
As a result, the guard at line 12 evaluates to true, and consequently the attacker succeeds in acquiring the access.

### 2.1.3 Alternative languages to C

Safe languages, such as Java and C#, are alternatives to C/C++ when security is a major concern. They use a combination of syntax restrictions, automatic memory management, and runtime checks to ensure memory safety and reduce memory errors. For example, Java’s syntax requires the function local variables to be initialized, so the code with any local variables declared but without initialization (as similar to the example of Figure 2.1) does not compile. In addition, when there is an out-of-bounds array access in Java code, the Java runtime would throw an exception, thus the attack in the example of Figure 2.2 cannot happen. Furthermore, Java’s garbage collection mechanism ensures implicit memory management for automatic heap de-allocations, therefore temporal errors do not exist any longer and it also helps reduce memory leaks to some extent in practice. As a result, undefined values and spatial/temporal safety violations are completely avoided when using safe languages.

Others, like Cyclone [39] and Deputy [14], manage to extend the original type system of C to guarantee memory safety. They often introduce some additional type information from programmer annotations, while preserving the low-level features of C.

Although these alternatives are effective in preventing memory errors, C/C++ is still commonly used in a wide range of today’s software. Operating systems, compilers, virtual machines, database management systems, web browsers, game engines, and some desktop applications and embedded software are typically written in C/C++. To prevent memory errors, a plausible solution is to port these
existing C/C++ programs to safe languages. Nevertheless, it is time-consuming and non-trivial to do so; and safe languages are not always appropriate for some of these specific domains. The trend of the widespread use of C/C++ in industry is likely to continue in at least the near future.

2.2 Detecting Memory Errors at Runtime

Basically, memory errors can be detected either statically or dynamically.

On one hand, static analysis approximates the possible runtime behaviors of a program without actually executing it. Due to its approximation nature, it is difficult to achieve both accuracy and scalability, especially for large programs. For example, it reports false positives.

On the other hand, dynamic analysis monitors the exact states during program execution, and performs checks to capture errors on the fly. The errors can be detected precisely in this manner, and thus runtime detection appears to be a more practical solution. For example, Memcheck [77] in Valgrind is widely used in industries; SoftBoundCETS [61, 62], AddressSanitizer [76], and MemorySanitizer [30] are adopted by the LLVM compiler infrastructure. The detailed techniques for dynamically detecting each type of memory errors are described individually in the rest of this section.

2.2.1 Detecting Undefined Value Uses

The basic idea to detect the uses of undefined values is to track the defindness of every value in the program and perform checks before potential safety violations. Existing work includes [7, 30, 36, 77]. Their fundamental idea is as follows:

- Every value is shadowed by a piece of metadata, which is used to record the
defindness of the value. The value can be in either register or memory. The metadata is usually implemented as a boolean value, indicating whether its corresponding original program value is properly defined or not.

- Every operation is shadowed by a *shadow operation* if it creates a new value (e.g., a binary operation). The shadow operation takes the metadata of the operands as inputs, and computes value for a new metadata (i.e., the metadata of the new value created by the original program operation). Thus, the definedness of every value in the program is propagated during the program execution.

- The definedness of every value is checked if it is used by an operation that could directly lead to safety violations. If any of the inputs taken by the operation is found to be undefined, the program execution is terminated with a warning reported.

In fact, the existence of undefined values is common in many programs, such as padding memory objects to enable compiler optimizations. It is also unnecessary and error-prone to check all operations, which may not cause a safety issue afterwards and thus a false alarm may occur. As a result, for the most part, the program tracks the definedness of every value; it performs checks for only a few operations that are potentially dangerous. Those operations may include conditional jumps, pointer dereferences, etc.

Although the shadow operations usually add checks for only a small portion of the overall operations, the underlying runtime overhead is still substantial due to the cost of shadow propagation for all values in the program.
2.2.2 Detecting Spatial Errors

The techniques to enforce spatial safety lies in the following three categories: guard zone-based approaches, object-based approaches, and pointer-based approaches. Each of them has its own advantages and disadvantages.

- **Guard Zone-based Approaches.** In guard zone-based approaches [35, 36, 66, 76, 107], every memory byte is marked as valid or invalid in the shadow memory, and the instrumentation checks the validity of memory locations accessed. Spatial safety violations are identified when accesses to invalid memory happen. In the memory layout organization, the valid memory objects are usually sandwiched between some special invalid memory chunks called *guard zones*, which must not be accessed by any spatial-safe program. The guard zones are also used to separate valid memory objects appropriately to make them sparsely allocated in the memory for space-efficient shadow mapping, and thus reduce the size of shadow memory.

- **Object-based Approaches.** In object-based approaches [1, 15, 17, 21, 41, 75], every memory object corresponds to its metadata indicating the bounds information. Such bounds information of an object is associated with the location of the object in memory. As a result, all pointers to an object share the same bounds information. On every pointer-manipulating operation, a spatial check is performed to ensure that the memory access is within the bounds of the same object. Usually, the range lookup is implemented as a splay tree.

- **Pointer-based Approaches.** In pointer-based approaches [2, 39, 61, 65, 70, 103], the bounds information is maintained per pointer (rather than per-object as in object-based approaches). Every pointer is associated with its
metadata, indicating the legal bounds of the object it points to. For every pointer dereference, a check is performed to determine if the memory region accessed is within the legal bounds. The metadata is generally placed adjacently with its corresponding pointer \cite{2, 39, 65, 70, 103}. The pointers with such inline metadata organization are referred as *fat-pointers*, which exhibit low source and binary compatibility since the memory layout of objects is changed. Recently, SoftBound \cite{61} has been proposed with a disjoint metadata scheme for improved compatibility.

In terms of compatibility, the pointer-based approaches are usually not compatible with un-instrumented libraries, which are pre-compiled without shadow operations. The pointers created by the libraries, as a result, miss their corresponding per-pointer metadata. On the contrary, guard zone-based and object-based approaches usually have better compatibility with un-instrumented libraries. The metadata associated with heap objects are properly updated by interpreting \texttt{malloc()} and \texttt{free()} function calls, even if the objects are allocated or de-allocated by un-instrumented code.

For soundness of error detection, pointer-based approaches ensure comprehensive spatial safety, while guard zone- and object-based approaches may miss some bugs. For guard zone-based approaches, in the case of overflows with a large stride that jumps over a guard zone and falls into another memory object, an out-of-bounds error will be missed. For object-based approaches, sub-object overflows (e.g., overflows of accesses to arrays inside structures) can not be detected.

### 2.2.3 Detecting the Other Errors

To enforce temporal safety, tools like Purify \cite{36}, Valgrind \cite{76}, Dr. Memory \cite{7}, and CETS \cite{62} maintain the lifetime status of memory objects in metadata. They
detect temporal errors by checking if the pointer used by a memory access or passed to the `free()` function points to a live memory object.

For memory leak detection, the instrumentation behaves similar as a mark-and-sweep garbage collector in [7, 77]. Such reachability-based analysis scans memory objects to identify those who no longer have any pointer pointing to them.

2.3 Program Instrumentation

Program instrumentation is a dynamic technique to monitor the real-time behavior of a system. It is implemented in the form of a set of program instructions inserted into the original program code with appropriate code organization. It can be used to measure code execution performance, record trace information, or diagnose errors at runtime. Generally speaking, the instrumentation code for a program can be inserted into either its binary or its source.

2.3.1 Binary-Level Instrumentation

Binary-level instrumentation inserts instructions for the binary code. Machine code is initially converted to a low-level IR; and the IR is then instrumented and transformed back to the target machine code. Examples of popular binary-level instrumentation frameworks include Valgrind [66], DynamoRIO [6, 112], and Pin [52].

Binary-level instrumentation provides good flexibility since it operates on binaries, and recompilation is never required. It is useful especially for the target programs whose source code is not accessible by users. However, it is not practical when performance is a major concern. For instance, programs instrumented using Memcheck [77] or Dr. Memory [7] are an order of magnitude slower than the native code.
2.3.2 Source-Level Instrumentation

Source-level instrumentation is mostly implemented at compile-time. The inserted code is IR-specific for different compilers used. It requires the source code of target programs; and even if the target program itself is open-source, it may still not be able to handle external functions from the libraries whose source code is not provided.

Compared to binary-level approaches, source-level instrumentation usually yields significantly better performance due to better register allocation, as well as a series of code optimizations performed at compile-time. To detect undefined value uses, a typical slowdown of MSan is 3X; for spatial error detection, SoftBound usually incurs a slowdown within 2X.

This thesis aims at efficient error detection, and thus uses source-level instrumentation techniques.

2.4 Background of LLVM

The Low-Level Virtual Machine (LLVM) compiler infrastructure was originally designed and developed by Lattner et al. [44]. It provides language- and platform-independent compilation based on its powerful and flexible intermediate representation (IR). It is written in C++ and can be used for program analysis, optimizations, and other modern compilation purposes.

This thesis chooses LLVM as a development foundation for the following reasons:

- the flexible IR for analysis and instrumentation;
- its robustness to compile common programs;
• some existing program analysis and transformation passes that can be lever-
aged;

• some state-of-the-art tools for dynamic memory error detection already avail-
able in LLVM (e.g., [30, 61]);

• the active community providing great support for development.

The rest of this section provides introduction to some LLVM background knowl-
edge that is relevant to this thesis.

2.4.1 Compiler Architecture

The open-source LLVM project, started in 2000 at the University of Illinois at
Urbana-Champaign as a research project, has now become an industrial-strength
platform. Together with the Clang front-end, it is competitive to those classic
C/C++ compilers, such as the GNU C Compiler (GCC) and Open64. Its highly
modular architecture with an efficient, easily maintainable, and reusable codebase
is a major benefit. As a result, it is now widely used in both academia and industry,
with rich and growing resources available.

LLVM uses a universal intermediate representation, namely LLVM-IR, for the
total compilation strategy, i.e., from front-end parsing all the way to target code
generation. Figure 2.3 shows a typical compilation workflow using the LLVM
toolchain. The source files are initially parsed by front-ends to generate LLVM
bitcode files (i.e., files with .bc extension), expressed in LLVM-IR. The bitcodes
are then individually optimized by LLVM passes, and linked into a single merged
bitcode file afterwards. Next, the optimizer runs again on this merged bitcode to
look for some extra optimization opportunities. Finally, the optimized bitcode is
passed to the target code generator and the system linker to produce an executable
LLVM is designed to be language- and system-independent due to its low-level IR. Apart from C/C++, it currently supports a range of other programming languages with appropriate front-ends, such as Fortran, OCaml, Haskell, Java bytecode, Scala, Objective-C, Swift, Python, Ruby, Go, Rust, etc. For machine code generation, it also covers a number of popular instruction sets, including X86/X86-64, ARM, MIPS, Nvidia PTX, PowerPC, etc.

![LLVM compilation toolchain diagram]

Figure 2.3: LLVM compilation toolchain.
2.4.2 LLVM-IR

The concept behind LLVM-IR is to make it low-level, typed, lightweight, and flexible for extensions. In LLVM-IR, a program is made up of one or several modules, where each module consists of a list of global variables and function definitions. Like most programming languages, a group of instructions are organized in a function. Instructions are generally performed with values (including an infinite amount of virtual registers), determining the program behaviors in details. Every value is associated with a type, thus some optimizations are directly allowed on the code without performing extra analysis.

LLVM’s instruction set is relatively simple, since it is designed to represent common operations. Machine instructions for specific targets are generated when LLVM-IR is lowered in the back end. The LLVM instructions related to this thesis are as follows:

- memory allocator *alloca* which creates a local stack variable and returns its address as a pointer value;
- memory access *load (store)* that reads from (writes to) a memory location via a pointer;
- pointer arithmetic *getelementptr* that gets the address of a subelement of a memory object;
- computational instructions including unary and binary instructions;
- jump instructions causing control flow transfers;
- function calls and returns.

LLVM-IR consists of two types of memory objects: (1) top-level and (2) address-taken variables. Top-level variables are the LLVM’s virtual registers which can be
accessed directly. Address-taken variables include global variables, heap allocations, and local variables that are not top-level variables (created by \texttt{alloca}). They can only be accessed indirectly via \texttt{loads} and \texttt{stores}, which transfer values between virtual registers and memory. An address-taken variable can only appear in a statement where its address is taken, and a top-level variable never has its address taken.

LLVM-IR appears in static single assignment (SSA) form for its top-level variables, i.e., every virtual register is written only once. However, it does not use the SSA form for address-taken variable representation. Leveraging pointer analysis, our study in Chapter 3 builds memory SSA based on LLVM-IR for value-flow analysis.

### 2.4.3 Some Relevant LLVM Passes

Our study leverages some existing LLVM passes, either the LLVM official ones or those provided by third parties. Some typical passes include:

- \textbf{mem2reg}. This LLVM’s built-in transformation pass promotes some local memory variables to virtual registers. It looks for \texttt{alloca}s which are used directly by loads and stores only, and promotes them by applying the standard \textit{iterated dominance frontier} algorithm. It does not handle structs and arrays; and other local variables, whose addresses are passed to a function or with pointer arithmetic involved, are not promoted either.

  The code is transformed in pruned SSA form, where the possible \texttt{alloca}s are promoted into SSA registers, with their corresponding loads and stores eliminated as appropriate and some necessary PHI nodes inserted. This is the foundation of many other analysis and optimization passes.
• **Scalar Evolution.** This analysis pass calculates closed-form expressions (SCEVs) for all top-level scalar integer variables. It abstracts a set of instructions that contribute to the value of a scalar into a single SCEV to focus on the overall calculation. Thus, it simplifies code analysis and optimizations for our study in Chapter 4.

• **Andersen.** This is an efficient implementation of the Andersen’s inclusion-based pointer analysis provided by Hardekopf and Lin [32, 33]. It is an interprocedural pass, which analyzes the entire program. For every pointer in the program, it conservatively computes the possible memory objects that can be pointed by this pointer. It serves as a pre-analysis phase for our Usher tool to compute value-flow graphs in Chapter 3.

• **MSan.** Implemented by Google, this transformation pass has been adopted by the official LLVM release since Version 3.3. It is a state-of-the-art tool to detect the uses of undefined values at runtime [30]. Our study in Chapter 3 involves it as the baseline; and the detailed techniques will be described in Chapter 3.

• **SoftBound.** This state-of-the-art tool that enforces spatial memory safety is provided by Nagarakatte et al. [61]. It is released as a part of the SoftBoundCETS open-source project [62], and is chosen as the baseline for our study in Chapter 4. The technical details about this tool will be discussed in Chapter 4.
2.5 Chapter Summary

In this chapter, we have introduced the memory errors of C programs, and reviewed the existing work of dynamic detection; we have also discussed the two types of program instrumentation techniques and presented some background about the LLVM compiler infrastructure, based on which our experiments were carried out. In the next two chapters, we will present our methodologies in accelerating the runtime detection of (1) uses of undefined values and (2) spatial errors, respectively.
Chapter 3

Accelerating Detection of Undefined Value Uses with Value-Flow Analysis

3.1 Overview

Uninitialized variables in C/C++ programs can cause system crashes if they are used in some critical operations (e.g., pointer dereferencing and branches) and security vulnerabilities if their contents are controlled by attackers. The undefinedness of a value can be propagated widely throughout a program directly (via assignments) or indirectly (via the results of operations using the value), making uses of undefined values hard to detect efficiently and precisely.

Static analysis tools [8, 40] can warn for the presence of uninitialized variables but usually suffer from a high false positive rate. As such, they typically sacrifice soundness (by missing bugs) for scalability in order to reduce excessively high false positives that would otherwise be reported.
To detect more precisely uses of undefined values (with fairly low false positives), dynamic analysis tools are often used in practice. During an instrumented program’s execution, every value is shadowed, and accordingly, every statement is also shadowed. For a value, its shadow value maintains its definedness to enable a runtime check to be performed on its use at a critical operation (Definition 1).

The instrumentation code for a program can be inserted into either its binary [7, 77] or its source [30, 36]. Binary instrumentation causes an order of magnitude slowdown (typically 10X - 20X). In contrast, source instrumentation can be significantly faster as it reaps the benefits of optimizations performed at compile time. For example, MSan (MemorySanitizer) [30], a state-of-the-art tool that adopts the latter approach, is reported to exhibit a typical slowdown of 3X but is still costly, especially for some programs.

Both approaches suffer from the problem of blindly performing shadow propagations for all the values and definedness checks at all the critical operations in a program. In practice, most values in real programs are defined. The shadow propagations and checks on a large percentage of these values can be eliminated since their definedness can be proved statically. In addition, a value that is never used at any critical operation does not need to be tracked.

In this chapter, we present a static value-flow analysis framework, called Usher, to accelerate uninitialized variable detection performed by source-level instrumentation tools such as MSan for C programs. We demonstrate its usefulness by evaluating an implementation in LLVM against MSan using all the 15 SPEC2000 C programs. Specifically, this chapter makes the following contributions:

• We introduce a new static value-flow analysis, Usher, for detecting uses of undefined values in C programs. Usher reasons about statically the definedness of values using a value-flow graph that captures def-use chains
for all variables interprocedurally and removes unnecessary instrumentation by solving a graph reachability problem. *Usher* is field-, flow- and context-sensitive wherever appropriate and supports two flavors of strong updates. Our value-flow analysis is sound (by missing no bugs statically) as long as the underlying pointer analysis is. This work represents the first such whole-program analysis for handling top-level and address-taken variables to guide dynamic instrumentation for C programs.

- We show that our VFG representation allows advanced instrumentation-reducing optimizations to be developed (with two demonstrated in this chapter). In addition, its precision can be improved orthogonally by leveraging existing and future advances on pointer analysis.

- We show that *Usher*, which is implemented in LLVM, can reduce the slowdown of MSAN from 212% – 302% to 123% – 140% for all the 15 SPEC2000 C programs under a number of configurations tested.

The rest of the chapter is organized as follows. Section 3.2 introduces a subset of C as the basis to present our techniques. Section 3.3 describes our *Usher* framework. Section 3.4 presents and analyzes our experimental results. Section 3.5 discusses the related work. Section 3.6 concludes.

## 3.2 Preliminaries

In Section 3.2.1, we introduce *TinyC*, a subset of C, to allow us to present our *Usher* framework succinctly. In Section 3.2.2, we highlight the performance penalties incurred by shadow-memory-based instrumentation.
Chapter 3. Accelerating Detection of Undefined Value Uses

3.2.1 TinyC

As shown in Figure 3.1, TinyC represents a subset of C. A program is a set of functions, with each comprising a list of statements (marked by labels from \(Lab\)) followed by a return. TinyC includes all kinds of statements that are sufficient to present our techniques: assignments, memory allocations, loads, stores, branches and calls. We distinguish two types of allocation statements, (1) \(x := \text{alloc}_\rho^T\), where the allocated memory \(\rho\) is initialized, and (2) \(x := \text{alloc}_\rho^F\), where the allocated memory \(\rho\) is not initialized.

Without loss of generality, we consider only local variables, which are divided into (1) the set \(\text{Var}^{TL}\) of top-level variables (accessed directly) and (2) the set \(\text{Var}^{AT}\) of address-taken variables (accessed indirectly only via top-level pointers). In addition, all variables in \(\text{Var}^{TL} \cup \text{Var}^{AT}\) and all constants in \(\text{Const}\) have the same type.

TinyC mimics LLVM-IR \[44\] in how the \& (address) operation as well as loads and stores are represented. In TinyC, as illustrated in Figure 3.2, \& is absent since the addresses of variables are taken by using \(\text{alloc}_\rho^T\) and \(\text{alloc}_\rho^F\) operations and the two operands at a load/store must be both top-level variables. In Figure 3.2(c), we have \(\text{Var}^{TL} = \{a, i, x, y\}\), \(\text{Var}^{AT} = \{b, c\}\) and \(\text{Const} = \{10\}\).

3.2.2 Shadow-Memory-based Instrumentation

When a program is fully instrumented with shadow memory \([7, 30, 36, 77]\), the definedness of every variable \(v\) in \(\text{Var}^{TL} \cup \text{Var}^{AT}\) is tracked by its shadow variable, \(v \in \{T, F\}\), of a Boolean type. All constant values in \(\text{Const}\) are defined (with \(T\)). Whether a variable is initialized with a defined value or not upon declaration depends on the default initialization rules given. In C programs, for example, global variables are default-initialized but local variables are not.
Chapter 3. Accelerating Detection of Undefined Value Uses

\[
P ::= F^+ \quad \text{(program)}
\]

\[
F ::= \text{def } f(a) \{ \ell : \text{stmt}; \text{ ret } r; \} \quad \text{(function)}
\]

\[
\text{stmt} ::= x := n \quad \text{(constant copy)}
\]

\[
\quad | x := y \quad \text{(variable copy)}
\]

\[
\quad | x := y \otimes z \quad \text{(binary operation)}
\]

\[
\quad | x := \text{alloc}_{\rho}^T \quad \text{(allocation with } \rho \text{ initialized)}
\]

\[
\quad | x := \text{alloc}_{\rho}^F \quad \text{(allocation with } \rho \text{ not initialized)}
\]

\[
\quad | x := *y \quad \text{(load)}
\]

\[
\quad | *x := y \quad \text{(store)}
\]

\[
\quad | x := f(y) \quad \text{(call)}
\]

\[
\quad | \text{if } x \text{ goto } \ell \quad \text{(branch)}
\]

\[
x, y, z, a, r \in \text{Var}^{TL} \quad \rho \in \text{Var}^{AT} \quad n \in \text{Const} \quad \ell \in \text{Lab}
\]

Figure 3.1: The TINYC source language.

As the results produced by statements may be tainted by the undefined values used, every statement \( s \) is also instrumented by its shadow, denoted \( \bar{s} \). For example, \( x := y \otimes z \) is instrumented by \( \bar{x} := y \otimes z \), which implies that \( \bar{x} := y \land \bar{z} \) is executed at run time to enable shadow propagations, where \( \land \) represents the Boolean \text{AND} operator.

**Definition 1 (Critical Operations)** An operation performed at a load, store or branch is a critical operation.

A runtime check is made for the use of a value at every critical operation. If its shadow is \( \mathcal{F} \), a warning is issued.
By indiscriminately tracking all values and propagating their shadow values across all statements in a program, full instrumentation can slow the program down significantly.

### 3.3 The Usher Framework

As shown in Figure 3.3, Usher, which is implemented in LLVM, comprises five phases (described below). In “Memory SSA Construction”, each function in a program is put in SSA (Static Single Assignment) form based on the pointer information available. In “Building VFG”, a VFG that connects def-use chains interprocedurally is built (flow-sensitively) with two flavors of strong updates being supported. In “Definedness Resolution”, the definedness of all values is statically resolved context-sensitively. In “Guided Instrumentation”, the instrumentation code required is generated, with strong updates performed to shadow values. This phase is regarded as the key contribution of this chapter. In “VFG-based Optimizations”, some VFG-based optimizations are applied to reduce instrumentation overhead further. Compared to full instrumentation, our guided instrumentation is more
lightweight.

Usher is sound as long as the underlying pointer analysis is. So no uses of undefined values will be missed. In addition to being flow- and context-sensitive, our value-flow analysis is also field-sensitive to obtain improved precision.

### 3.3.1 Memory SSA Construction

Initially, Usher puts all functions in a program in SSA form, an IR where a variable is statically defined exactly once. In TinyC (as in LLVM-IR), def-use information for top-level variables is immediately available. However, def-use information for address-taken variables requires pointer analysis to discover how they are accessed.
Chapter 3. Accelerating Detection of Undefined Value Uses

\[ F ::= \text{def } f(a \ [\overline{x}])\{\ldots \text{ret } r \ [\overline{y}];\} \quad \text{(virtual input and output parameters)} \]

\[ \text{stmt} ::= \ldots \]

\[ | \quad x := \text{alloc}_\rho \ [\rho := \chi(\rho)] \quad \text{(allocation)} \]

\[ | \quad x := *y \ [\mu(\rho)] \quad \text{(load)} \]

\[ | \quad *x := y \ [\rho := \chi(\rho)] \quad \text{(store)} \]

\[ | \quad x \ [\overline{y}] := f(y \ [\overline{p}]) \quad \text{(call)} \]

\[ | \quad v := \phi(v, v) \quad \text{(phi)} \]

\[ v \in \text{Var}^{TL} \cup \text{Var}^{AT} \]

Figure 3.4: The TINYC language in SSA form.

indirectly as \textit{indirect defs} at stores and \textit{indirect uses} at loads.

Figure 3.4 shows how TINYC is extended to allow a TINYC program to be put in SSA form. Note that \(\phi\) is the standard function for handling control-flow join points. Following [12], we use \(\mu\) and \(\chi\) functions to, respectively, indicate the potentially indirect uses and defs of address-taken variables at loads, stores and allocation sites. Each load \(x := *y\) is annotated with a list \(\overline{\mu(\rho)}\) of \(\mu\) functions, where each \(\mu(\rho^k)\) function represents potentially an indirect use of \(\rho^k\) (that may be pointed to by \(y\)). Similarly, each store \(*x := y\) is annotated with a list \(\overline{\rho := \chi(\rho)}\) of \(\chi\) functions, where each \(\rho^k := \chi(\rho^k)\) function represents potentially an indirect use and def of \(\rho^k\) (that may be pointed to by \(x\)). At an allocation site, a single \(\rho := \chi(\rho)\) function is added, where \(\rho\) is the name of the address-taken variable allocated.

A function def \(f(a) \{\ldots, \text{ret } r;\}\) is extended to make explicit (1) all address-taken variables (called \textit{virtual formal parameters}) that are used, i.e., read in \(f\)
... 
\texttt{a := alloc}_{r}^{F}; \quad \cdots \quad \texttt{a}_{1} := \text{alloc}_{b}^{F} \quad [b_{2} := \chi(b_{1})];
\texttt{. . . := foo(a);} \quad \cdots \quad \texttt{. . . := foo(a_{1}[b_{2}]);}
\texttt{. . .}
\texttt{. . .}
\texttt{. . .}
\texttt{. . .}
\texttt{\textbf{def foo(q) \{}} \quad \texttt{\textbf{def foo(q}_{1}[b_{1}]) \{}}
\texttt{x := *q;} \quad \texttt{x}_{1} := *q_{1} \quad [\mu(b_{1})];
\texttt{if x goto 1;} \quad \texttt{if x}_{1} \texttt{goto 1}';
\texttt{t := 10;} \quad \texttt{t}_{1} := 10;
\texttt{x := x \odot t;} \quad \texttt{x}_{2} := x_{1} \odot t_{1};
\texttt{*q := x;} \quad \texttt{*q}_{1} := x_{2} \quad [b_{2} := \chi(b_{1})];
\texttt{1 : ret x;} \quad \texttt{1' : x}_{3} := \phi(x_{1}, x_{2});
\texttt{}} \quad \texttt{b}_{3} := \phi(b_{1}, b_{2});
\texttt{}} \quad \texttt{ret x_{3}[b_{3}];}
\texttt{}}

\begin{figure}[h]
\begin{center}
(a) TINYC \hspace{2cm} (b) SSA
\end{center}
\end{figure}

Figure 3.5: A TINYC program and its SSA form.

directly or indirectly via \(a\), and (2) all address-taken variables (called \textit{virtual output parameters}) that are either modified in \(f\) via \(a\) or returned by \(r\), directly or indirectly. Accordingly, the syntax for the call sites of \(f\) is extended. For a function \(f\) and its call sites, \(\rho^{k}\) (the \(k\)-th element) in each of the \(\overline{\rho}\) lists used always represents the same address-taken variable.

Once all required \(\mu\) and \(\chi\) functions have been added, every function is put in SSA form individually by using a standard SSA construction algorithm. Figure 3.5 gives an example. It is understood that different occurrences of a variable with the same version (e.g., \(b_{1}\) and \(b_{2}\)) are different if they appear in different functions. Recall that each \(\rho := \chi(\rho)\) function represents a potential use and def of \(\rho\) \cite{12}. In \(b_{2} := \chi(b_{1})\) associated with \(*q_{1} := x_{2}\), \(b_{1}\) indicates a potential use of the previous definition of \(b\) and \(b_{2}\) a potentially subsequent re-definition of \(b\). The opportunities for strong updates at a \(\chi\) function are explored below.
3.3.2 Building Value-Flow Graph

During this phase, Usher builds a value-flow graph for a program to capture the def-use chains both within a function and across the function boundaries in a program. What is novel about this phase is that two types of strong updates are considered for store statements.

For each definition \( v_r \) in the SSA form of a program, where \( r \) is the version of \( v \), we write \( \hat{v}_r \) for its node in the VFG. We sometimes elide the version number when the context is clear. A value-flow edge \( \hat{v}_m \leftarrow \hat{v}_n \) indicates a data dependence of \( v_m \) on \( v_n \). Since we are only concerned with checking the definedness of a value used at a critical operation, it suffices to build the VFG only for the part of the program dependent by all critical operations.

For an allocation site \( x_r := \text{alloc}_I [\rho_m := \chi(\rho_n)] \), where \( I \in \{T, F\} \), we add \( \hat{x}_r \leftarrow \hat{T} \) (since \( x_r \) points to \( \rho \) ), \( \hat{\rho}_m := \hat{T} \) and \( \hat{\rho}_m \leftarrow \hat{\rho}_n \). Here, \( \hat{T} \) and \( \hat{F} \) are two special nodes, called the root nodes in the VFG, with \( \hat{T} \) representing a defined value and \( \hat{F} \) an undefined value.

For an assignment representing a copy, binary operation, load or \( \phi \) statement of the form \( x_m := \ldots \), we add \( \hat{x}_m \leftarrow \hat{y}_n \) for every use of \( y_n \) on the right-hand side of the assignment. Given \( a_2 := b_3 \odot c_4 \), for example, \( \hat{a}_2 \leftarrow \hat{b}_3 \) and \( \hat{a}_2 \leftarrow \hat{c}_4 \) will be added. Given \( d_4 := 10 \), \( \hat{d}_4 \leftarrow \hat{T} \) will be created.

For stores, we consider both traditional strong and weak updates as well as a new semi-strong update. Consider a store \(*x_s = y_t [\rho_m := \chi(\rho_n)]\). If \( x_s \) uniquely points to a concrete location \( \rho \), \( \rho_m \) can be strongly updated. In this case, \( \rho_m \) receives whatever \( y_t \) contains and the value flow from \( \rho_n \) is killed. So only \( \hat{\rho}_m \leftarrow \hat{y}_t \) is added. Otherwise, \( \rho_m \) must incorporate the value flow from \( \rho_n \), by also including \( \hat{\rho}_m \leftarrow \hat{\rho}_n \). As a result, \( \rho_m \) can only be weakly updated.

Presently, Usher uses a pointer analysis that does not
Chapter 3. Accelerating Detection of Undefined Value Uses

```python
def foo(q1[b1]) {
    b2 := ϕ(b1, b4);
    q1 := alloc[b] [b3 := χ(b2)];
    p1 := q1;
    t1 := 0;
    *p1 := t1 [b4 := χ(b3)];
    ... if ... goto 1;
    ... ret ...;
}
```

(a) TinyC (b) VFG

Figure 3.6: A semi-strong update performed at \(*p1 := t1\). With a weak update, \(B_4 \leftarrow B_3\) would be introduced. With a semi-strong update, this edge is replaced (indicated by a cross) by \(B_4 \leftarrow B_2\) (indicated by the dashed arrow) so that \(B_3 \leftarrow \sim\) is bypassed.

provide must-alias information. We improve precision by also performing a semi-strong update for a store \(*x_s := y_t\) \([\rho_m := \chi(\rho_n)]\), particularly when it resides in a loop. Suppose there is an allocation site \(z_r := \text{alloc}_{\rho} [\cdot := \chi(\rho_j)]\) such that \(\hat{z}_r\) dominates \(\hat{x}_s\) in the VFG, which implies that \(z_r := \text{alloc}_{\rho} \) dominates \(*x_s := y_t\) in the CFG (Control-Flow Graph) of the program as both \(z_r\) and \(x_s\) are top-level variables. This means that \(x_s\) uniquely points to \(\rho\) created at the allocation site. Instead of adding \(\hat{\rho}_m \leftarrow \hat{y}_t\) and \(\hat{\rho}_m \leftarrow \hat{\rho}_n\) by performing a weak update, we will add \(\hat{\rho}_m \leftarrow \hat{y}_t\) and \(\hat{\rho}_m \leftarrow \hat{\rho}_j\).

Consider an example given in Figure 3.6, where \(\text{foo}\) may be called multiple times so that the address-taken variable \(b\) is both used (read) and modified inside. At the store \(*p1 := t1\) \(p1\) points to an abstract location. So a strong update is impossible. If a weak update is applied, \(B_4 \leftarrow \hat{t}_1\) and \(B_3 \leftarrow \hat{B}_3\) will be introduced, causing USHER to conclude that \(b_4\) may be undefined due to the presence of \(B_3 \leftarrow \sim\). Since
\( q_1 \) dominates \( p_1 \), a semi-strong update can be performed at the store \( *p_1 := t_1 \).

Instead of \( b_4 \leftarrow b_3 \), which is introduced by a weak update, \( b_4 \leftarrow b_2 \) is added, so that \( b_3 \leftarrow \hat{F} \) will be bypassed. Usher can then more precisely deduce that \( b_4 \) is defined as long as \( b_1 \) is.

Finally, we discuss how to add value-flow edges across the function boundaries. Consider a function definition
\[
def f(a_1[p_1^1, p_1^2, \ldots]) \{ \ldots \text{ret} r_s[p_1^1, p_2^1, \ldots]; \},
\]
where \( p_k^1 (p_k^r) \) is the \( k \)-th virtual input (output) parameter with version 1 \( (i_k) \). For each call site \( x_t[p_1^1, p_2^2, \ldots] = f(y_m[p_1^1, p_2^2, \ldots]) \), we add \( \hat{a}_t \leftarrow \hat{y}_m \) and \( \hat{r}^k_s \leftarrow \hat{r}^k_{h_k} \) (for every \( k \)) to connect each actual argument to its corresponding formal parameter. Similarly, we also propagate each output parameter to the call site where it is visible, by adding \( \hat{x}_t \leftarrow \hat{r}_s \) and \( \hat{r}_s^k \leftarrow \hat{r}^k_{h_k} \) (for every \( k \)).

### 3.3.3 Definedness Resolution

Presently, Usher instruments every function in a program only once (without cloning the function). Therefore, the definedness of all the variables (i.e., values) in the VFG of the program can be approximated by a graph reachability analysis, context-sensitively by matching call and return edges to rule out unrealizable interprocedural flows of values in the standard manner [51, 73, 79, 89, 102].

Let \( \Gamma \) be a function mapping the set of nodes in the VFG to \( \{ \bot, \top \} \). The definedness, i.e., state of a node \( \hat{v} \) is \( \Gamma(\hat{v}) = \bot \) if it is reachable by the root \( \hat{F} \) and \( \Gamma(\hat{v}) = \top \) otherwise (i.e., if it is reachable only by the other root, \( \hat{F} \)).

### 3.3.4 Guided Instrumentation

Instead of shadowing all variables and statements in a program, Usher solves a graph reachability problem on its VFG by identifying only a subset of these to be
instrumented at run time. The instrumentation code generated by USHER is sound as long as the underlying pointer analysis used is. This ensures that all possible undefined values flowing into every critical operation in a program are tracked at run time.

During this fourth phase (and also the last phase in Section 3.3.5), USHER works on a program in SSA form. To avoid cluttering, we often refer to an SSA variable with its version being elided since it is deducible from the context.

A statement may need to be shadowed only if the value $\hat{v}$ defined (directly/indirectly) by the statement can reach a node $\hat{x}$ that satisfies $\Gamma(\hat{x}) = \bot$ in the VFG such that $x$ is used in a critical statement. A sound instrumentation implies that all shadow values accessed by any shadow statement at run time are well-defined.

Given a statement $\ell : s$, we formally define an instrumentation item for $\ell : s$ as a pair $\langle \ell , s \rangle$ or $\langle \ell , \hat{s} \rangle$, indicating that the shadow operation (or statement) $\hat{s}$ for $s$ is inserted just before or after $\ell$ (with $s$ omitted). The instrumentation item sets for different types of statements are computed according to the instrumentation rules given in Figures 3.7 – 3.9.

The deduction rules are formulated in terms of

$$\mathcal{P}, \Gamma \vdash \hat{\nu} \downarrow \Sigma_{\nu} \quad (3.1)$$

where $\Sigma_{\nu}$ is the set of instrumentation items that enables the flows of undefined values into node $\hat{\nu}$ to be tracked soundly via shadow propagations. This is achieved by propagating the $\Sigma$’s of $\hat{\nu}$’s predecessors in the VFG into $\hat{\nu}$ and also adding relevant new instrumentation items for $\hat{\nu}$. Here, $\mathcal{P}$ is a given program in SSA form. In addition, $\mathcal{P}(code)$ holds if the block of statements, denoted code, exists in $\mathcal{P}$.

In shadow-memory-based instrumentation, a runtime shadow map, denoted $\sigma$, is maintained for mapping variables (or precisely their locations) to (the locations
of) their shadow variables. \( \sigma_g \) is a global variable introduced at run time to shadow parameter passing. In addition, \( E \) records at run time whether a critical statement has accessed an undefined value or not.

The guided instrumentation for \( P \) is precisely specified as the union of \( \Sigma \)'s computed by applying the rules in Figures 3.7 – 3.9 to all nodes representing the uses at critical operations. In \( \text{T-Check} \) and \( \text{L-Check} \), \( \hat{\ell} \) denotes a virtual node (due to the existence of a virtual assignment of the form \( \ell := x \)) associated with the critical statement \( \ell \) to ease the presentation.

Different propagation schemes are used for \( \text{T-nodes} \) \( \hat{v} \) (where \( \Gamma(\hat{v}) = T \) and

\[
\begin{align*}
\text{[T-Check]} & \quad s \in \{ \_ := *x, *x := \_ \text{ if } x \text{ goto } \_ \} \quad P(\ell : s) \quad \Gamma(\hat{x}) = T \\
& \quad P, \Gamma \vdash \hat{\ell} \downarrow \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{[T-Assign]} & \quad s \in \{ x := n/y, x := \text{alloc}_\_ \text{, } x := \_ \otimes \_ \text{, } x := *\_ \text{, } x [\_] := f(\_\_) \} \\
& \quad P(\ell : s) \quad \Gamma(\hat{x}) = T
\end{align*}
\]

\[
\begin{align*}
\text{[T-Para]} & \quad P(\text{def } f(a [\_\])\{\ell : \_ \text{ ; } \ldots \}) \quad \Gamma(\hat{a}) = T \\
& \quad P, \Gamma \vdash \hat{a} \downarrow \{ \langle \ell, \sigma(a) := T \rangle \}
\end{align*}
\]

\[
\begin{align*}
\text{[T-Alloc]} & \quad P(\ell : x := \text{alloc}_\rho^T [\rho_m := \chi(\_\_)]) \quad \Gamma(\hat{\rho_m}) = T \\
& \quad P, \Gamma \vdash \hat{\rho_m} \downarrow \{ \langle \ell, \sigma(*) := T \rangle \}
\end{align*}
\]

\[
\begin{align*}
\text{[T-Store}^\text{SU}] & \quad P(\ell : *x := \_ [\rho_m := \chi(\_\_)]) \quad \Gamma(\hat{\rho_m}) = T \\
& \quad P, \Gamma \vdash \hat{\rho_m} \downarrow \{ \langle \ell, \sigma(*) := T \rangle \}
\end{align*}
\]

\[
\begin{align*}
\text{[T-Store}^\text{WU/SemiSU}] & \quad P(\_ : *\_ := \_ [\_ , \rho_m := \chi(\_\_), \_]) \quad \Gamma(\hat{\rho_m}) = T \\
& \quad \Gamma, \hat{\rho_m} \dashv \hat{\rho_n} \\
& \quad P, \Gamma \vdash \hat{\rho_m} \downarrow \Sigma \hat{\rho_m}
\end{align*}
\]

Figure 3.7: Instrumentation rules for \( \text{T-nodes} \).
Chapter 3. Accelerating Detection of Undefined Value Uses

\begin{align*}
\text{[\bot-Check]} & \quad s \in \{\_, := *x, *x := \_, \text{ if } x \text{ goto } \_\} \quad \begin{array}{c}
\mathcal{P}(\ell : s) \\
\Gamma(\widehat{x}) = \bot
\end{array} \quad \mathcal{P}, \Gamma \vdash \widehat{x} \downarrow \Sigma_x \\
\quad \mathcal{P}, \Gamma \vdash \widehat{\ell} \downarrow \Sigma_x \cup \{(\widehat{\ell}, \mathcal{E}(\ell) := (\sigma(x) = \mathcal{F}))\}
\end{align*}

\begin{align*}
\text{[\bot-VCopy]} & \quad \mathcal{P}(\ell : x := y) \\
\Gamma(\widehat{x}) = \bot \\
\mathcal{P}, \Gamma \vdash \widehat{y} \downarrow \Sigma_y \\
\mathcal{P}, \Gamma \vdash \widehat{x} \downarrow \Sigma_x \cup \{(\widehat{\ell}, \sigma(x) := \sigma(y))\}
\end{align*}

\begin{align*}
\text{[\bot-Bop]} & \quad \mathcal{P}(\ell : x := y \otimes z) \\
\mathcal{P}, \Gamma \vdash \widehat{y} \downarrow \Sigma_y \\
\mathcal{P}, \Gamma \vdash \widehat{z} \downarrow \Sigma_z \\
\mathcal{P}, \Gamma \vdash \widehat{x} \downarrow \Sigma_x \cup \{(\widehat{\ell}, \sigma(x) := \sigma(y) \land \sigma(z))\}
\end{align*}

\begin{align*}
\text{[\bot-Alloc]} & \quad \mathcal{P}(\ell : x := \text{alloc}^\rho y / \text{alloc}^\rho \rho_m) [\rho_m := \chi(\rho_n)] \\
\Gamma(\widehat{\rho}_m) = \bot \\
\mathcal{P}, \Gamma \vdash \widehat{\rho}_n \downarrow \Sigma_{\widehat{\rho}_n} \\
\mathcal{P}, \Gamma \vdash \widehat{\rho}_m \downarrow \Sigma_{\widehat{\rho}_m} \cup \{(\widehat{\ell}, \sigma(*x) := [\mathcal{F} / \mathcal{F}])\}
\end{align*}

\begin{align*}
\text{[\bot-Para]} & \quad \mathcal{P}(\text{def } f(a [\_])\{\ell : \_; \ldots \}) \\
\quad \mathcal{C}_f := \{\ell_i \mid \ell_i \text{ is a call site for function } f\} \\
\quad \forall \ell_i \in \mathcal{C}_f, \mathcal{P}(\ell_i : x := f(y^i [\_])); \mathcal{P}, \Gamma \vdash \widehat{y}^i \downarrow \Sigma_{\widehat{y}} \\
\mathcal{P}, \Gamma \vdash \widehat{x} \downarrow (\bigcup_i \Sigma_{\widehat{y}}) \cup \{(\widehat{\ell}, \sigma(a) := \sigma_g), (\widehat{\ell_i}, \sigma_g := \sigma(y^i))\}
\end{align*}

\begin{align*}
\text{[\bot-Ret]} & \quad \mathcal{P}(\text{def } f(\_); \ldots \ell' : \_; \text{ ret } r [\_];) \\
\quad \mathcal{P}(\ell : x [\_] := f(\_)) \\
\quad \Gamma(\widehat{x}) = \bot \\
\quad \mathcal{P}, \Gamma \vdash \widehat{r} \downarrow \Sigma_{\widehat{r}} \\
\mathcal{P}, \Gamma \vdash \widehat{x} \downarrow \Sigma_{\widehat{x}} \cup \{(\widehat{\ell}, \sigma(x) := \sigma_g), (\widehat{\ell'}, \sigma_g := \sigma(r))\}
\end{align*}

\begin{align*}
\text{[\bot-Load]} & \quad \forall \rho^i : \mathcal{P}, \Gamma \vdash \widehat{\rho}_i \downarrow \Sigma_{\widehat{\rho}_i} \\
\quad \Gamma(\widehat{x}) = \bot \\
\mathcal{P}, \Gamma \vdash \widehat{x} \downarrow (\bigcup_i \Sigma_{\widehat{\rho}}) \cup \{(\widehat{\ell}, \sigma(x) := \sigma(*y))\}
\end{align*}

\begin{align*}
\text{[\bot-Store]_{\text{SU/WU/SemiSU}}} & \quad \mathcal{P}(\ell : *x := y [\_, \rho_m := \chi(\_, \_)] \\
\Gamma(\widehat{\rho}_m) = \bot \\
\mathcal{P}, \Gamma \vdash \widehat{y} \downarrow \Sigma_y \\
\mathcal{P}, \Gamma \vdash \widehat{x} \downarrow \Sigma_x \\
\mathcal{P}, \Gamma \vdash \widehat{\rho}_m \downarrow \Sigma_{\widehat{\rho}_m} \cup \{(\widehat{\ell}, \sigma(*x) := \sigma(y))\} \cup \Sigma_{\widehat{\rho}_m}
\end{align*}

Figure 3.8: Instrumentation rules for \bot-nodes.
Chapter 3. Accelerating Detection of Undefined Value Uses

\[ \Phi \]  
\[ \text{P}(\vdash v_i := \phi(v_m, v_n)) \quad \text{P}, \Gamma \vdash \widehat{v_m} \downarrow \Sigma_{\widehat{v_m}} \quad \text{P}, \Gamma \vdash \widehat{v_n} \downarrow \Sigma_{\widehat{v_n}} \]

\[ \text{VPara} \]  
\[ \text{P}(\text{def } f(\ldots \rho_m, \ldots )) \quad \forall \widehat{\rho}_m \leftarrow \widehat{\rho}_i : \text{P}, \Gamma \vdash \widehat{\rho}_i \downarrow \Sigma_{\widehat{\rho}_i} \]

\[ \text{VRet} \]  
\[ \text{P}(\vdash \rho_m.r := f(\ldots )) \quad \text{P}, \Gamma \vdash \widehat{\rho}_m \downarrow \Sigma_{\widehat{\rho}_m} \]

Figure 3.9: Instrumentation rules for virtual nodes.

\( \bot \)-nodes \( \widehat{v} \) (where \( \Gamma(\widehat{v}) = \bot \)). The rules are divided into three sections (separated by the dashed lines): (1) those prefixed by \( \top \) for \( \top \)-nodes, (2) those prefixed by \( \bot \) for \( \bot \)-nodes, and (3) the rest for some “virtual” nodes introduced for handling control-flow splits and joins.

Special attention should be paid to the rules (that apply to \( \top \)-nodes only), where a shadow location can be strongly updated. The remaining rules are straightforward. Consider a statement where \( \sigma(v) \) needs to be computed for a variable \( v \) at run time. We say that \( \sigma(v) \) can be strongly updated if \( \sigma(v) := \top \) can be set directly at run time to indicate that \( v \) is defined at that point so that the (direct or indirect) predecessors of \( \widehat{v} \) in the VFG do not have to be instrumented with respect to \( v \) at this particular statement.

\( \top \)-Nodes  Let us first consider the rules for \( \top \)-nodes. The value flow of a (top-level or address-taken) variable \( v \) is mimicked exactly by that of its shadow \( \sigma(v) \). There are two cases in which a strong update to \( \sigma(v) \) can be safely performed. For top-level variables, this happens in \( [\top \text{-Assign}] \) and \( [\top \text{-Para}] \), which are straightforward to understand.

For address-taken variables, strong updates are performed in \( [\top \text{-Alloc}] \) and \( [\top \text{-Store}^{\text{SU}}] \) but not in \( [\top \text{-Store}^{\text{WU/semiSU}}] \). For an allocation site \( x := \)}
Chapter 3. Accelerating Detection of Undefined Value Uses

$$\text{alloc}^\mathcal{T}_\rho [\rho_m := \chi(x)],$$ such that $$\Gamma(\widehat{\rho}_m) = \top,$$ \(*x\) uniquely represents the location $$\rho_m,$$ which contains a well-defined value. Therefore, $$\sigma(*x)$$ can be strongly updated, by setting $$\sigma(*x) := \mathcal{T}$$ $$([\top-\text{Alloc}]).$$

Let us consider an indirect def $$\rho_m$$ at a store, where $$\widehat{\rho}_m$$ is a $$\top$$-node. As discussed in Section 3.3.2, $$\widehat{\rho}_m$$ has at most two predecessors. One predecessor represents the variable, say $$y_t$$, on the right-hand side of the store. The shadow propagation for $$y_t$$ is not needed since $$\Gamma(\widehat{\rho}_m) = \top$$ implies $$\Gamma(\widehat{y}_t) = \top.$$ The other predecessor represents an older version of $$\rho,$$ denoted $$\rho_n.$$ If $$\widehat{\rho}_m \leftarrow \widehat{\rho}_n$$ is absent, then $$[\top-\text{Store}^\mathcal{SU}]$$ applies. Otherwise, $$[\top-\text{Store}^\mathcal{WU/SemiSU}]$$ applies. In the former case, $$\sigma(*x) := \mathcal{T}$$ is strongly updated as $$x$$ uniquely points to a concrete location $$\rho.$$ However, the same cannot happen in $$[\top-\text{Store}^\mathcal{WU/SemiSU}]$$ since the resulting instrumentation would be incorrect otherwise. Consider the following code snippet:

\begin{verbatim}
*p_2 := t_1 [b_3 := \chi(b_2), c_4 := \chi(c_3)];
... := *q_3 [\mu(b_3);
\end{verbatim}

Even $$\Gamma(b_3) = \Gamma(c_4) = \top,$$ we cannot directly set $$\sigma(*p) := \mathcal{T}$$ due to the absence of strong updates to $$b$$ and $$c$$ at the store. During a particular execution, it is possible that $$p_2$$ points to $$c$$ but $$q_3$$ points to $$b.$$ In this case, $$*p_2$$ is not a definition for $$b.$$ If $$b$$ needs to be shadowed at the load, its shadow $$\sigma(b)$$ must be properly initialized earlier and propagated across the store to ensure its well-definedness at the load.

Finally, a runtime check is not needed at a critical operation when a defined value is used $$([\top-\text{Check}]).$$

$$\bot-\text{Nodes}$$ Now let us discuss the rules for $$\bot$$-nodes. The instrumentation code is generated as in full instrumentation, requiring the instrumentation items for its predecessors to be generated to enable shadow propagations into this node. $$[\bot-\text{VCopy}]$$ and $$[\bot-\text{Bop})$$ are straightforward to understand. For an allocation site $$x :=
alloc$^T_{\rho}(\text{alloc}^F_{\rho})[\rho_m \leftarrow \chi(\rho_n)]$, such that $\Gamma(\hat{\rho}_m) = \perp, \sigma(*x)$, i.e., the shadow for the object currently allocated at the site, is strongly updated to be $T(F)$. In addition, the older version $\rho_n$ is tracked as well.

The standard parameter passing for a function is instrumented so that the value of the shadow of its actual argument at every call site is propagated into the shadow of the (corresponding) formal parameter ($\perp$-Para). This is achieved by using an auxiliary global variable $\sigma_g$ to relay an shadow value across two different scopes. Retrieving a value returned from a function is handled similarly ($\perp$-Ret)

At a load $x := *y$, where $\Gamma(\hat{x}) = \perp$, all the indirect uses made via $*y$ must be tracked separately to enable the shadow propagation $\sigma(x) := \sigma(*y)$ for the load ($\perp$-Load).

In $\perp$-Store$^\text{SU/WU/SemiSU}$, strong updates to shadow locations cannot be safely performed. In particular, the value flow from the right-hand side $y$ of a store must also be tracked, unlike in $T$-Store$^\text{SU}$ and $T$-Store$^\text{WU/SemiSU}$.

When an undefined value $x$ may be potentially used at a critical statement at $\ell$, a runtime check must be performed at the statement ($\perp$-Check). In this case, $E(\ell)$ is set to true if and only if $\sigma(x)$ evaluates to $F$.

**Virtual Nodes** For the “virtual” value-flow edges added due to $\phi$ and parameter passing for virtual input and output parameters, the instrumentation items required will be simply collected across the edges, captured by $[\Phi], [\text{VPara}]$ and $[\text{VRet}]$. During program execution, the corresponding shadow values will “flow” across such value-flow edges.
3.3.5 VFG-based Optimizations

Our VFG representation is general as it allows various instrumentation-reducing optimizations to be developed. Below we describe two optimizations, developed based on the concept of Must Flow-from Closure (MFC), denoted $\nabla$.

**Definition 2 (MFC)** $\nabla_x$ for a top-level variable $x$ is:

\[
\nabla_x := \begin{cases} 
\{\hat{x}\} \cup \nabla_{\hat{y}} \cup \nabla_{\hat{z}}, & \mathcal{P}(x := y \otimes z) \\
\{x\} \cup \nabla_{\hat{y}}, & \mathcal{P}(x := y) \\
\{\hat{x}, \hat{f}\}, & \mathcal{P}(x := n) \text{ or } \mathcal{P}(x := \text{alloc}) \\
\{\hat{x}\}, & \text{otherwise}
\end{cases}
\]

It is easy to see that $\nabla_x$ is a DAG (directed acyclic graph), with $\hat{x}$ as the (sole) sink and one or more sources (i.e., the nodes without incoming edges). In addition, $\Gamma(\hat{x}) = \top$ if and only if $\Gamma(\hat{y}) = \top$ for all nodes $\hat{y}$ in $\nabla_x$.

$\nabla_x$ contains only top-level variables because loads and stores cannot be bypassed during shadow propagations.

**Optimization I: Value-Flow Simplification**

This optimization (referred to as Opt I later) aims to reduce shadow propagations in an MFC. For each $\nabla_x$, the shadow value $\sigma(x)$ of a top-level variable $x$ is a conjunct of the shadow values of its source nodes. Thus, it suffices to propagate directly the shadow values of the sources $s$, such that $\Gamma(\hat{s}) = \bot$, to $\hat{x}$, as illustrated in Figure 3.10.

**Optimization II: Redundant Check Elimination**

Our second optimization (Opt II) is more elaborate but also conceptually simple. The key motivation is to reduce instrumentation overhead by avoiding spurious
\[ x_1 := a_1 \land b_1; \\
\]
\[ y_1 := c_1 \land d_1; \\
\]
\[ z_1 := x_1 \lor y_1; \\
\]
\[ \ldots \]

Figure 3.10: An example of value-flow simplification.

\[ c_1 := a_1 \land b_1; \\
\]
\[ l_1 : \ldots := *c_1 [\ldots]; \\
\]
\[ \ldots \]
\[ d_1 := 0; \\
\]
\[ e_1 := b_1 \land d_1; \\
\]
\[ l_2 : \text{if } e_1 \text{ goto } \ldots; \\
\]
\[ \ldots \]

Figure 3.11: An example for illustrating redundant check elimination, where \( l_1 \) is assumed to dominate \( l_2 \) in the CFG of the program. If \( b_1 \) has an undefined value, then the error can be detected at both \( l_1 \) and \( l_2 \). The check at \( l_2 \) can therefore be disabled by a simple modification of the original VFG.
Algorithm 1 Redundant Check Elimination

begin
1 \( G \leftarrow \text{the VFG of the program } P; \)
2 \( \text{foreach top-level variable } x \in \text{Var}^{TL} \text{ used at a critical statement, denoted } s, \text{ in } P \text{ do} \)
3 \( \nabla_{\hat{x}} \leftarrow \text{MFC computed for } \hat{x} \text{ in } G; \)
4 \( \nabla'_{\hat{x}} \leftarrow \nabla_{\hat{x}} \cup \{ \rho_m \mid \hat{y} \in \nabla_{\hat{x}}, \ P(y := *z \ [\mu(\rho_m)]), \rho_m \in \text{Var}^{AT} \text{ represents a} \)
5 \( \text{concrete location}\}; \)
6 \( \mathcal{R}_{\hat{x}} \leftarrow \{ \hat{r} \mid \hat{t} \in \nabla'_{\hat{x}}, \hat{r} \notin \nabla'_{\hat{x}}, \hat{r} \leftarrow \hat{t} \text{ in } G\}; \)
7 \( \text{foreach statement } s_r, \text{ where } \hat{r} \in \mathcal{R}_{\hat{x}} \text{ is defined do} \)
8 \( \text{if } s \text{ dominates } s_r \text{ in the CFG of } P \text{ then} \)
9 \( \text{Replace every } \hat{r} \leftarrow \hat{t}, \text{ where } \hat{t} \in \nabla'_{\hat{x}}, \text{ by } \hat{r} \leftarrow \hat{T} \text{ in } G; \)
10 \( \text{Perform definedness resolution to obtain } \Gamma \text{ on } G; \)

Figure 3.11(c) modified from Figure 3.11(b), by replacing \( \hat{e}_1 \leftarrow \hat{b}_1 \) with \( \hat{e}_1 \leftarrow \hat{T} \), then no runtime check at \( l_2 \) is necessary (since \([\top\text{-Check}] \) is applicable to \( l_2 \) when \( \Gamma(e_1) = \top \)).

As shown in Algorithm 1, we perform this optimization by modifying the VFG of a program and then recomputing \( \Gamma \). If an undefined value \textit{definitely} flows into a critical statement \( s \) via either a top-level variable in \( \nabla_{\hat{x}} \) or possibly an address-taken variable \( \rho_m \) (lines 3 – 4), then the flow of this undefined value into another node \( \hat{r} \) outside \( \nabla_{\hat{x}} \) (lines 5 – 6) such that \( s \) dominates \( s_r \), where \( \hat{r} \) is defined, can be redirected from \( \hat{T} \) (lines 7 – 8). As some value flows from address-taken variables may have been cut (line 9), \textsc{Usher} must perform its guided instrumentation on the VFG (obtained without this optimization) by using \( \Gamma \) obtained here to ensure that all shadow values are correctly initialized.
3.4 Evaluation

The main objective is to demonstrate that by performing a value-flow analysis, Usher can significantly reduce instrumentation overhead of MSan, a state-of-the-art source-level instrumentation tool for detecting uses of undefined values.

3.4.1 Implementation

We have implemented Usher in LLVM (version 3.3), where MSan is released. Usher uses MSan’s masked offset-based shadow memory scheme for instrumentation and its runtime library to summarize the side effects of external functions on the shadow memory used.

Usher performs an interprocedural whole-program analysis to reduce instrumentation costs. All source files of a program are compiled and then merged into one bitcode file (using LLVM-link). The merged bitcode is transformed by iteratively inlining the functions with at least one function pointer argument to simplify the call graph (excluding those functions that are directly recursive). Then LLVM’s mem2reg is applied to promote memory into (virtual) registers, i.e., generate SSA for top-level local variables. We refer to this optimization setting as O0+IM (i.e., LLVM’s O0 followed by Inlining and Mem2reg). Finally, LLVM’s LTO (Link-Time Optimization) is applied.

For the pointer analysis phase shown in Figure 3.3, we have used an offset-based field-sensitive Andersen’s pointer analysis [32]. Arrays are treated as a whole. 1-callsite-sensitive heap cloning is applied to allocation wrapper functions. 1-callsite context-sensitivity is configured for definedness resolution (Section 3.3.3). In addition, access-equivalent VFG nodes are merged by using the technique from [34].

In LLVM, all the global variables are accessed indirectly (via loads and stores)
and are thus dealt with exactly as address-taken variables. Their value flows across
the function boundaries are realized as virtual parameters as described in Figure 3.4
and captured by $[\text{VPara}]$ and $[\text{VRet}]$.

Like MSAN, Usher’s dynamic detection is bit-level precise [77], for three rea-
sons. First, Usher’s static analysis is conservative for bit-exactness. Second, at
run time, every bit is shadowed and the shadow computations for bit operations
in $[\text{\texttt{Bop}}]$ (defined in Figure 3.8) are implemented as described in [77]. Finally, $\nabla_x$
given in Definition 2 is modified so that $P(x := y \oplus z)$ holds when $\oplus$ is not a bitwise
operation.

### 3.4.2 Platform and Benchmarks

All experiments are done on a machine equipped with a 3.00GHz quad-core In-
tel Core2 Extreme X9650 CPU and 8GB DDR2 RAM, running a 64-bit Ubuntu
10.10. All the 15 C benchmarks from SPEC CPU2000 are used and executed under
their reference inputs. Some of their salient properties are given in Table 3.1 and
Table 3.2, with explanations below.

### 3.4.3 Methodology

Like MSAN, Usher is designed to facilitate detection of uninitialized variables.
O0+IM represents an excellent setting for obtaining meaningful stack traces in
error messages. In addition, LLVM under “-O1” or higher flags behaves non-
deterministically on undefined (i.e., $\text{undef}$) values [111], making their runtime
detection nondeterministic. Thus, we will focus on comparing MSAN and Usher
under O0+IM in terms of instrumentation overhead when both are implemented
identically in LLVM except that their degrees of instrumentation differ. We will
examine both briefly in Section 3.4.6 when higher optimization flags are used.
In addition, we will also highlight the importance of statically analyzing the value flows for address-taken variables and evaluate the benefits of our VFG-based optimizations.

### 3.4.4 Value-Flow Analysis

We now analyze the results of value-flow analysis performed under O0+IM.

Table 3.1 presents the performance results of Usher’s value-flow analysis. Usher is reasonably lightweight, consuming under 10 seconds (inclusive pointer analysis time) and 600 MB memory on average. The two worst performers are 176.gcc and 253.perlbmk, both taking nearly 1 minute and consuming ≈2.7 and ≈1.4 GB memory, respectively. The latter is more costly when compared to other

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size (KLOC)</th>
<th>Time (secs)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>8.6</td>
<td>0.32</td>
<td>294</td>
</tr>
<tr>
<td>175.vpr</td>
<td>17.8</td>
<td>0.54</td>
<td>306</td>
</tr>
<tr>
<td>176.gcc</td>
<td>230.4</td>
<td>58.35</td>
<td>2,758</td>
</tr>
<tr>
<td>177.mesa</td>
<td>61.3</td>
<td>1.88</td>
<td>366</td>
</tr>
<tr>
<td>179.art</td>
<td>1.2</td>
<td>0.28</td>
<td>291</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2.5</td>
<td>0.28</td>
<td>292</td>
</tr>
<tr>
<td>183.equake</td>
<td>1.5</td>
<td>0.29</td>
<td>293</td>
</tr>
<tr>
<td>186.crafty</td>
<td>21.2</td>
<td>0.70</td>
<td>315</td>
</tr>
<tr>
<td>188.ammp</td>
<td>13.4</td>
<td>0.57</td>
<td>307</td>
</tr>
<tr>
<td>197.parser</td>
<td>11.4</td>
<td>0.79</td>
<td>315</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>87.1</td>
<td>53.93</td>
<td>1,405</td>
</tr>
<tr>
<td>254.gap</td>
<td>71.5</td>
<td>19.21</td>
<td>701</td>
</tr>
<tr>
<td>255.vortex</td>
<td>67.3</td>
<td>11.15</td>
<td>601</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4.7</td>
<td>0.30</td>
<td>293</td>
</tr>
<tr>
<td>300.twolf</td>
<td>20.5</td>
<td>1.26</td>
<td>331</td>
</tr>
<tr>
<td>average</td>
<td>41.4</td>
<td>9.99</td>
<td>591</td>
</tr>
</tbody>
</table>

Table 3.1: Performance of Usher’s value-flow analysis.
benchmarks with similar sizes, since its larger VFG contains more interprocedural value-flow edges for its global and heap variables, which are both in $\text{Var}^{AT}$.

In Table 3.2, some statistics for both $\text{Var}^{TL}$ (containing the virtual registers produced by mem2reg) and $\text{Var}^{AT}$ are given for each benchmark. In LLVM, global variables belong to $\text{Var}^{AT}$ and are accessed via loads and stores. This explains why all benchmarks except 255. vortex have more global variables than stack variables (that are not converted to virtual registers by mem2reg). However, at an allocation site $x := \text{alloc}^{\rho}$, where $\rho$ is a global variable, $x$ is a $\text{const}$ top-level pointer and is thus always initialized ($\text{\lbrack T-Alloc\rbrack}$). So it needs not to be checked when used at a critical statement. In the last column (under “$\%F$”), we see that 34% of the

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$\text{Var}^{TL}$ (10$^3$)</th>
<th>$\text{Var}^{AT}$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack</td>
<td>Heap</td>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>164.gzip</td>
<td>7</td>
<td>27</td>
<td>10</td>
<td>428</td>
</tr>
<tr>
<td>175.vpr</td>
<td>22</td>
<td>177</td>
<td>207</td>
<td>770</td>
</tr>
<tr>
<td>176.gcc</td>
<td>324</td>
<td>1,600</td>
<td>874</td>
<td>6,824</td>
</tr>
<tr>
<td>177.mesa</td>
<td>113</td>
<td>738</td>
<td>2,417</td>
<td>2,534</td>
</tr>
<tr>
<td>179.art</td>
<td>2</td>
<td>8</td>
<td>48</td>
<td>83</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2</td>
<td>8</td>
<td>89</td>
<td>71</td>
</tr>
<tr>
<td>183.equake</td>
<td>4</td>
<td>32</td>
<td>29</td>
<td>122</td>
</tr>
<tr>
<td>186.crafty</td>
<td>29</td>
<td>71</td>
<td>528</td>
<td>1,460</td>
</tr>
<tr>
<td>188.ammp</td>
<td>26</td>
<td>76</td>
<td>342</td>
<td>416</td>
</tr>
<tr>
<td>197.parser</td>
<td>16</td>
<td>184</td>
<td>447</td>
<td>1,005</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>116</td>
<td>736</td>
<td>814</td>
<td>3,705</td>
</tr>
<tr>
<td>254.gap</td>
<td>125</td>
<td>54</td>
<td>4,101</td>
<td>4,313</td>
</tr>
<tr>
<td>255.vortex</td>
<td>76</td>
<td>3,576</td>
<td>1,548</td>
<td>3,602</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>5</td>
<td>21</td>
<td>13</td>
<td>166</td>
</tr>
<tr>
<td>300.twolf</td>
<td>52</td>
<td>116</td>
<td>700</td>
<td>841</td>
</tr>
<tr>
<td>average</td>
<td>61</td>
<td>495</td>
<td>811</td>
<td>1,756</td>
</tr>
</tbody>
</table>

Table 3.2: Variable statistics. “$\%F$” is the percentage of address-taken variables uninitialized when allocated.
address-taken variables are not initialized when allocated on average. Note that heap objects allocated at a `calloc()` site or its wrappers are always initialized (`[T-Alloc]`).

Table 3.3 shows the information of different types of updates performed on stores. In Columns 3, we can see some good opportunities for traditional strong updates, which kill undefined values to enable more $T$-nodes to be discovered statically. According to the pointer analysis used [32], at 82% of the stores (on average), a (top-level) variable in $\text{Var}^{TL}$ points to one single abstract object in $\text{Var}^{AT}$, with 82% being split into 36%, where strong updates are performed, and 46%, where

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Stores</th>
<th>%SU</th>
<th>%WU*</th>
<th>$\bar{S}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>617</td>
<td>62</td>
<td>34</td>
<td>-</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1,044</td>
<td>34</td>
<td>53</td>
<td>1.2</td>
</tr>
<tr>
<td>176.gcc</td>
<td>10,851</td>
<td>40</td>
<td>31</td>
<td>4.7</td>
</tr>
<tr>
<td>177.mesa</td>
<td>7,798</td>
<td>6</td>
<td>63</td>
<td>0.2</td>
</tr>
<tr>
<td>179.art</td>
<td>140</td>
<td>41</td>
<td>59</td>
<td>-</td>
</tr>
<tr>
<td>181.mcf</td>
<td>221</td>
<td>25</td>
<td>70</td>
<td>-</td>
</tr>
<tr>
<td>183.equake</td>
<td>189</td>
<td>26</td>
<td>68</td>
<td>-</td>
</tr>
<tr>
<td>186.crafty</td>
<td>2,215</td>
<td>63</td>
<td>28</td>
<td>-</td>
</tr>
<tr>
<td>188.ammp</td>
<td>1,291</td>
<td>11</td>
<td>76</td>
<td>4.9</td>
</tr>
<tr>
<td>197.parser</td>
<td>892</td>
<td>34</td>
<td>60</td>
<td>2.9</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>8,904</td>
<td>52</td>
<td>11</td>
<td>5.7</td>
</tr>
<tr>
<td>254.gap</td>
<td>4,378</td>
<td>16</td>
<td>28</td>
<td>-</td>
</tr>
<tr>
<td>255.vortex</td>
<td>6,169</td>
<td>70</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>303</td>
<td>32</td>
<td>68</td>
<td>-</td>
</tr>
<tr>
<td>300.twolf</td>
<td>2,989</td>
<td>34</td>
<td>38</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Table 3.3: Updates performed on stores. “%SU” is the percentage of stores with strong updates. “%WU*” is the percentage of stores $x = y$ with $x$ pointing to one address-taken variable (where weak stores would be performed if semi-strong updates are not applied). “$\bar{S}$” is the number of times our semi-strong update rule is applied per non-array heap allocation site.
Chapter 3. Accelerating Detection of Undefined Value Uses

weak updates would have to be applied. In the last column, we see that the average number of times that our semi-strong update rule (introduced in Section 3.3.2) is applied, i.e., the average number of cuts made on the VFGs (highlighted by a cross in Figure 3.6) per non-array heap allocation site is 3.2.

The statistics for value-flow graph of each benchmark are listed in Table 3.4. By performing static analysis, Usher can avoid shadowing the statements that never produce any values consumed at a critical statement, where a runtime check is needed. Among all the VFG nodes (Column 2), only an average of 38% may need to be tracked (Column 3). In the second last column, the average number of simplified MFCs (Definition 2) by Opt I is 15251. In the last column, the average

| Benchmark     | #Nodes ($10^3$) | %|B| | $S_\nabla$ ($10^3$) | |R| ($10^3$) |
|---------------|-----------------|---|---|----------------|---|---|
| 164.gzip      | 16              | 20 | 0.6 | 1.0            |   |
| 175.vpr       | 51              | 27 | 3.2 | 4.9            |   |
| 176.gcc       | 17,932          | 56 | 96.0 | 54.3          |   |
| 177.mesa      | 151             | 22 | 8.7 | 15.8          |   |
| 179.art       | 5               | 21 | 0.2 | 0.6            |   |
| 181.mcf       | 4               | 4  | 0.0 | 0.7            |   |
| 183.equake    | 6               | 11 | 0.6 | 0.9            |   |
| 186.crafty    | 103             | 34 | 2.1 | 4.5            |   |
| 188.ammp      | 55              | 32 | 4.7 | 6.7            |   |
| 197.parser    | 162             | 81 | 1.9 | 3.3            |   |
| 253.perlbmk   | 8,378           | 84 | 41.9 | 23.2          |   |
| 254.gap       | 1,941           | 48 | 49.5 | 21.8          |   |
| 255.vortex    | 2,483           | 78 | 7.7 | 11.6          |   |
| 256.bzip2     | 11              | 16 | 0.3 | 1.2            |   |
| 300.twolf     | 122             | 37 | 11.2 | 12.4          |   |
| average       | 2,095           | 38 | 15.3 | 10.9          |   |

Table 3.4: Value-flow graph statistics. “%|B|” is the percentage of the VFG nodes reaching at least one critical statement, where a runtime check is needed. “$S_\nabla$” stands for the number of $\nabla$’s simplified by Opt I. “|R|” is the size of the union of $R_{\tilde{x}}$’s for all $\tilde{x}$ defined in line 5 of Algorithm 1 by Opt II.
number of VFG nodes connected to $\overline{r}$ by Opt II, as illustrated in Figure 3.11, is 10859.

### 3.4.5 Instrumentation Overhead

Figure 3.12 compares Usher and MSan in terms of their relative slowdowns to the native (instrumentation-free) code for the 15 C benchmarks tested. MSan has an average slowdown of 302%, reaching 493% for 253.perlbmk. With guided instrumentation, Usher has reduced MSan’s average slowdown to 123%, with 340% for 253.perlbmk. In addition, we have also evaluated three variations of Usher: (1) Usher$^{TL}$, which analyzes top-level variables only without performing Opt I and Opt II, which are described in Section 3.3.5, (2) Usher$^{TL+AT}$, which is Usher$^{TL}$ extended to handle also address-taken variables, and (3) Usher$^{OptI}$, which is Usher$^{TL+AT}$ extended to perform Opt I only. The average slowdowns for Usher$^{TL}$, Usher$^{TL+AT}$ and Usher$^{OptI}$ are 272%, 193% and 181%, respectively. One use of an undefined value is detected in the function $ppmatch()$ of 197.parser by all the analysis tools.

Figure 3.13 shows the static number of shadow propagations (i.e., reads from shadow variables) and the static number of runtime checks (at critical operations) performed by the four versions of our analysis (normalized with respect to MSan). Usher$^{TL}$ can remove 43% of all shadow propagations and 28% of all checks performed by MSan, reducing its slowdown from 302% to 272%. By analyzing also address-taken variables, Usher$^{TL+AT}$ has lowered this slowdown more visibly to 193%, by eliminating two-thirds of the shadow propagations and more than half of the checks performed by MSan. This suggests that a sophisticated value-flow analysis is needed to reduce unnecessary instrumentation for pointer-related operations. There are two major benefits. First, the flows of defined values from
Figure 3.12: Execution time slowdowns (normalized with respect to native code).
Figure 3.13: Static numbers of shadow propagations and checks performed at critical operations (normalized with respect to MSAN).
address-taken variables are now captured statically. Second, the statements that contribute no value flow to a critical operation do not need to be instrumented at all. However, the performance differences between \textsc{Usher}^{TL} and \textsc{Usher}^{TL+AT} are small for 253.perlbmk and 254.gap. For 253.perlbmk, the majority (84\%) of its VFG nodes reach a critical statement, where a runtime check is needed, as shown in Table 3.4. For 254.gap, there are a high percentage (49\%) of uninitialized address-taken variables when allocated and a relatively small number of strong updates (at 16\%).

The two VFG-based optimizations bring further benefits to the \textsc{Usher} framework. Compared to \textsc{Usher}^{TL+AT}, \textsc{Usher}^{OptI} requires fewer shadow propagations, down from 32\% to 22\% on average, causing the slowdown to drop from 193\% to 181\%. If Opt II is also included, \textsc{Usher} can lower further the number of shadow propagations from 22\% to 16\% and the number of checks from 44\% to 23\%, resulting in an average slowdown of 123\%. Due to Opt II, more nodes (10859 on average) are connected with \(\tilde{T}\), as shown in Figure 3.11. In an extreme case, 181.mcf suffers from only a 2\% slowdown. In this case, many variables that are used at frequently executed critical statements have received \(T\).

### 3.4.6 Effect of Compiler Optimizations on Reducing Instrumentation Overhead

We have also compared \textsc{Usher} and \textsc{MSAN} under their respective higher optimization settings, O1 and O2, even though this gives LLVM an opportunity to hide some uses of undefined values counter-productively \cite{111}, as discussed earlier. For an optimization level (O1 or O2) under both tools, a source file is optimized by (1) performing the LLVM optimizations at that level, (2) applying the \textsc{Usher} or \textsc{MSAN} analysis to insert the instrumentation code, and (3) rerunning the optimiza-
tion suite at that level to further optimize the instrumentation code inserted.

MSAN and Usher suffer from 231% and 140% slowdowns, respectively, under O1, and 212% and 132%, respectively, under O2 on average. The best performer for both tools is 164.gzip, with 104% (O1) and 102% (O2) for MSAN, and 26% (O1) and 20% (O2) for Usher. 255.vortex is the worst performer for MSAN, with 501% (O1) and 469% (O2), and also for Usher under O1 with 300%. However, the worst performer for Usher under O2 is 253.perlbmk with 288%. Note that Usher has higher slowdowns under O1 and O2 than O0+IM, since the base native programs benefit relatively more than instrumented programs under the higher optimization levels (in terms of execution times).

Therefore, Usher has reduced MSAN’s instrumentation costs by 39.4% (O1) and 37.7% (O2) on average. Compared to O0+IM, at which Usher achieves an overhead reduction of 59.3% on average, the performance gaps have been narrowed when advanced compiler optimizations are enabled.

The users can choose different configurations to suit their different needs. For analysis performance, they may opt to O1 or O2 at the risk of missing bugs and having to decipher mysterious error messages generated. For debugging purposes, they should choose O0+IM.

### 3.5 Related Work

#### 3.5.1 Detecting Uses of Undefined Values

As previously introduced in Section 2.2, prior studies rely mostly on dynamic instrumentation (at the binary- or source-level). The most widely used tool, Memcheck [77], was developed based on the Valgrind runtime instrumentation framework [66]. Recently, Dr. Memory [7], which is implemented on top of DynamoRIO
[6, 112], runs twice as fast as Memcheck but is still an order of magnitude slower than the native code, although they both detect other bugs besides undefined memory uses. A few source-level instrumentation tools are also available, including Purify [36] and MSan [30]. Source-level instrumentation can reap the benefits of compile-time optimizations, making it possible for MSAN to achieve a typical slowdown of 3X.

There are also some efforts focused on static detection [8, 40]. In addition, GCC and LLVM’s clang can flag usage of uninitialized variables. However, their analysis are performed intraprocedurally, leading to false positives and false negatives. The problem of detecting uses of undefined values can also be solved by traditional static analysis techniques, including IFDS [73], typestate verification [22] and type systems [65] (requiring source-code modifications). However, due to its approximate nature, static analysis alone finds it rather difficult to maintain both precision and efficiency.

### 3.5.2 Combining Static and Dynamic Analysis

How to combine static and dynamic analysis has been studied for a variety of purposes. On one hand, static analysis can guide dynamic analysis to reduce its instrumentation overhead. Examples include taint analysis [11], buffer overflow attack protection [13], detection of other memory corruption errors [37] and WCET evaluation [56]. On the other hand, some concrete information about a program can be obtained at run time to improve the precision of static analysis. In [93], profiling information is used to guide source-level instrumentation by adding hooks to the identified contentious code regions to guarantee QoS in a multiple workload environment. In [82], dynamic analysis results are used to partition a streaming application into subgraphs, so that the static optimizations that are not scalable
for the whole program can be applied to all subgraphs individually.

To detect uses of undefined values, a few attempts have been made. In [67], compile-time analysis and instrumentation are combined to analyze array-based Fortran programs, at 5X slowdown. Their static analysis is concerned with analyzing the definedness of arrays by performing a data-flow analysis interprocedurally. In [65], the proposed approach infers the definedness of pointers in C programs and checks those uncertain ones at run time. However, manual source code modification is required to satisfy its type system.

### 3.5.3 Value-Flow Analysis

Unlike data-flow analysis, value-flow analysis computes the def-use chains relevant to a client and puts them in some sparse representation. This requires the pointer/alias information to be made available by pointer analysis. Some recent studies improve precision by tracking value flows in pointer analysis [34, 48, 49], memory leak detection [89], program slicing [84] and interprocedural SSA analysis [10].

### 3.5.4 Pointer Analysis

Although orthogonal to this work, pointer analysis can affect the effectiveness of our value-flow analysis. In the current implementation of Usher, the VFG of a program is built based on the pointer information produced by an offset-based field-sensitive Andersen’s pointer analysis available in LLVM [32]. To track the flow of values as precisely as possible, our value-flow analysis is interprocedurally flow-sensitive and context-sensitive. However, the presence of some spurious value-flow edges can reduce the chances for shadow values to be strongly updated. In addition, our context-sensitive definedness resolution may traverse some spurious value-flow paths unnecessarily, affecting its efficiency. So both the precision and
efficiency of our value-flow analysis can be improved by using more precise pointer analysis [34, 42, 46, 48, 49, 51, 78, 79, 80, 83, 88, 91, 92, 95, 101, 106, 109] in future.

3.6 Chapter Summary

This chapter introduces a new VFG-based static analysis, Usher, to speed up the dynamic detection of uses of undefined values in C programs. We have formalized and developed the first value-flow analysis framework that supports two flavors of strong updates to guide source-level instrumentation. Validation in LLVM using all the 15 SPEC2000 C programs demonstrates its effectiveness in significantly reducing the instrumentation overhead incurred by a state-of-the-art source-level dynamic analysis tool. The work proposed in this chapter was previously published in [105].
Chapter 4

Accelerating Enforcement of Spatial Memory Safety with Weakest Preconditions

4.1 Overview

C, together with its OO incarnation C++, is the de facto standard for implementing systems software (e.g., operating systems and language runtimes), embedded software as well as server and client applications. Due to the low-level control provided over memory allocation and layout, software written in such languages makes up the majority of performance-critical code running on most platforms. Unfortunately, these unsafe language features often lead to memory corruption errors, including spatial errors (e.g., buffer overflows) and temporal errors (e.g., use-after-free), causing program crashes and security vulnerabilities in today’s commercial software.

This chapter focuses on eliminating spatial errors, which directly result in
out-of-bounds memory accesses of all sort and buffer overflow vulnerabilities, for C. As a long-standing problem, buffer overflows remain to be one of the highly ranked vulnerabilities, as revealed in Figure 4.1 with the data taken from the NVD database [64]. In addition, a recent study shows that buffer overflows are the commonest vulnerability in the last quarter century [108]. Furthermore, spatial errors persist today, as demonstrated by a recently reported Heartbleed vulnerability in OpenSSL (CVE-2014-0160).

Several approaches exist for detecting and eliminating spatial errors for C/C++ programs at runtime: guard zone-based [35, 36, 66, 76, 107], object-based (by maintaining per-object bounds metadata) [1, 15, 17, 21, 41, 75], pointer-based (by maintaining per-pointer metadata) either inline [2, 39, 65, 70, 103] or in a disjoint shadow space [16, 28, 59, 61]. These approaches can be implemented in software via instrumentation, at source-level as in [21, 61, 76] or binary-level as
As no suggested hardware support is available yet, the software industry typically employs software-only approaches to enforce spatial safety.

Detecting spatial errors at runtime via instrumentation is conceptually simple but can be computationally costly. A program is instrumented with shadow code, which records and propagates bounds metadata and performs out-of-bounds checking whenever a pointer is used to access memory, i.e., dereferenced at a load \( \cdots = \ast p \) or a store \( \ast p = \cdots \). Such bounds checking can be a major source of runtime overheads, particularly if it is done inside loops or recursive functions.

Performing bounds checking efficiently is significant as it helps improve code coverage of a spatial-error detection tool. By being able to test against a larger set of program inputs (due to reduced runtime overheads), more input-specific spatial errors can be detected and eliminated. To this end, both software- and hardware-based optimizations have been discussed before. For example, a simple dominator-based redundant check elimination \([61]\) enables the compiler to avoid the redundant checks at any dominated memory accesses. As described in \([60]\) and also in the recently announced MPX ISA extensions from Intel \([38]\), new instructions are proposed to be added for accelerating bounds checking (and propagation).

In this chapter, we present a new compile-time optimization that not only complements prior bounds checking optimizations but also applies to any aforementioned spatial-error detection approach (in software or hardware or both). Based on the notion of Weakest Precondition (WP), its novelty lies in guarding a bounds check at a pointer dereference inside a loop, where the WP-based guard is hoisted outside the loop, so that its falsehood implies the absence of out-of-bounds errors at the dereference, thereby avoiding the corresponding bounds check inside the loop. In addition, a simple value-range analysis allows multiple memory
accesses to share a common guard, reducing further the associated bounds checking overheads. Finally, we apply loop unswitching to a loop to trade code size for performance so that some bounds checking operations in some versions of the loop are completely eliminated.

We demonstrate the effectiveness of our WP-based optimization by taking SoftBound [61] as the baseline. SoftBound, with an open-source implementation available in LLVM, represents a state-of-the-art compile-time tool for detecting spatial errors. By adopting a pointer-based checking approach with disjoint metadata, SoftBound provides source compatibility and completeness when enforcing spatial safety for C. By performing instrumentation at source-level instead of binary-level as in MemCheck [66], SoftBound can reduce MemCheck’s overheads significantly as both the original and instrumentation code can be optimised together by the compiler. However, SoftBound can still be costly, with its overheads reaching or exceeding 2X for some programs.

To boost the performance of SoftBound, we have developed a new tool, called WPBound, that is a refined version of SoftBound, also in LLVM, by incorporating our WP-based optimization. WPBound supports separate compilation since its analysis and transformation phases are intraprocedural. Our evaluation shows that WPBound is effective in reducing SoftBound’s instrumentation overheads while incurring some small code size increases.

In summary, the contributions of this chapter are:

• a WP-based optimization for reducing bounds checking overheads for C programs;

• a WP-based source-level instrumentation tool, WPBound, for enforcing spatial safety for C programs;
Chapter 4. Accelerating Detection of Spatial Errors

• an implementation of WPBOUND in LLVM;

• an evaluation on a set of 12 C programs, showing that WPBOUND reduces SOFTBOUND’s average runtime overhead from 71% to 45% (by a reduction of 37%), with small code size increases.

The rest of this chapter is organized as follows. Section 4.2 provides the background for this work. Section 4.3 motivates and describes our WP-based instrumentation approach. Section 4.4 evaluates and analyzes our approach. Section 4.5 discusses additional related work and Section 4.6 concludes.

4.2 Background

We review briefly how SOFTBOUND [61] works as a pointer-based approach. Section 4.5 discusses additional related work on guard zone- and object-based approaches in detail.

Figure 4.2 illustrates the pointer-based metadata initialization, propagation and checking abstractly in SOFTBOUND with the instrumentation code highlighted in orange. Instead of maintaining the per-pointer metadata (i.e., base and bound) inline [2, 39, 65, 70, 103], SOFTBOUND uses a disjoint metadata space to achieve source compatibility.

The bounds metadata are associated with a pointer whenever a pointer is created (Figure 4.2(a)). The types of base and bound are typically as char* so that spatial errors can be detected at the granularity of bytes. These metadata are propagated on pointer-manipulating operations such as copying and pointer arithmetic (Figure 4.2(b)).

When pointers are used to access memory, i.e., dereferenced at loads or stores, spatial checks are performed (Figures 4.2(c) and (d)) by invoking the sChk function.
int a;
int *p = &a;
char *p_bs = p, *p_bd = (char*)(p + 1);
float *q = malloc(n);
char *q_bs = q;
char *q_bd = (q == 0) ? 0 : (char*)q + n;

(a) Memory allocation

int *p, *q;
char *p_bs = 0, *p_bd = 0;
char *q_bs = 0, *q_bd = 0;
...
p = q; // p = q + i; (p = &q[i];)
p_bs = q_bs;
p_bd = q_bd;

(b) Copying and pointer arithmetic

float *p;
char *p_bs = 0, *p_bd = 0;
...
sChk(p, p_bs, p_bd, sizeof(float));
... = *p; // *p = ...;

(c) Scalar loads and stores

int **p, *q;
char *p_bs = 0, *p_bd = 0;
char *q_bs = 0, *q_bd = 0;
...
sChk(p, p_bs, p_bd, sizeof(int*));
q = *p; // *p = q;
q_bs = GM[p]->bs; // GM[p]->bs = q_bs;
q_bd = GM[p]->bd; // GM[p]->bd = q_bd;

(d) Pointer loads and Stores

inline void sChk(char *p, char *p_bs, char *p_bd, size_t size) {
    if (p < p_bs || p + size > p_bd) {
        ... // Issue an error message.
        abort();
    }
}

(e) Spatial checks

Figure 4.2: Pointer-based instrumentation with disjoint metadata.
shown in Figures 4.2(e). The base and bound of a pointer is available in a disjoint shadow space and can be looked up in a global map $GM$. $GM$ can be implemented in various ways, including a hash table or a trie. For each spatial check, five x86 instructions, `cmp`, `br`, `lea`, `cmp`, and `br`, are executed on x86, incurring a large amount of runtime overheads, which will be significantly reduced in our WPBOUND framework.

To detect and prevent out-of-bounds errors at a load $\cdots = *p$ or a store $*p = \cdots$, two cases are distinguished depending on whether $*p$ is a scalar pointer (Figure 4.2(c)) or a non-scalar pointer (Figure 4.2(d)). In the latter case, the metadata for the pointer $*p$ (i.e., the pointer pointed by $p$) in $GM$ is retrieved for a load $\cdots = *p$ and updated for a store $*p = \cdots$.

### 4.3 The WPBOUND Framework

WPBOUND, which is implemented in the LLVM compiler infrastructure, consists of one analysis and three transformation phases (as shown in Figure 4.3). Their functionalities are briefly described below, illustrated by an example in Section 4.3.1, and further explained in Sections 4.3.3 and 4.3.4. As its four phases are intraprocedural, WPBOUND provides transparent support for separate compilation.

**Value Range Analysis** This analysis phase computes conservatively the value ranges of pointers dereferenced at loads and stores, leveraging LLVM’s *scalar evolution pass*. The value range information is used for the WP computations in the following three transformation phases, where the instrumentation code is generated.

**Loop-Directed WP Abstraction** This phase inserts spatial checks for memory accesses (at loads and stores). For each access in a loop, we reduce its bounds
checking overhead by exploiting but not actually computing exactly the WP that verifies the assertion that an out-of-bounds error definitely occurs at the access during some program execution. As value-range analysis is imprecise, a WP is estimated conservatively, i.e., weakened. For convenience, such WP estimates are still referred to as WPs. For each access in a loop, its bounds check is guarded by its WP, with its evaluation hoisted outside the loop, so that its falsehood implies the absence of out-of-bounds errors at the access, causing its check to be avoided.

**WP Consolidation** As an optimization, this phase consolidates the WPs for multiple accesses, which are always made to the same object, into a single one.

**WP-Driven Loop Unswitching** As another optimization that trades code size for performance, loop unswitching is applied to a loop so that the instrumen-
tation in its frequently executed versions is effectively eliminated.

4.3.1 A Motivating Example

We explain how WPBOUND works with a program in C given in Figure 4.4 (rather its LLVM low-level code).

```c
#define S sizeof(int)
#define SP sizeof(int*)
...
int i, k, L;
int t1, t2;
int *a, *b;
int **p;
...
if(...)
{
    sChk(p+k, p_bs, p_bd, SP);
    a = p[k];
}

sChk(p+k+1, p_bs, p_bd, SP);
b = p[k+1];
...
for(i = 1; i <= L; i++)
{
    sChk(a+i-1, a_bs, a_bd, S);
    t1 = a[i-1];
    if(t < ...)
    {
        sChk(b+i, b_bs, b_bd, S);
        t2 = b[i];
        sChk(a+i, a_bs, a_bd, S);
        a[i] += t1 + t2;
    }
}
```

(a) Unoptimized instrumentation

```c
inline bool wpChk(char *p_lb, char *p_ub,
                   char *p_bs, char *p_bd) {
    return p_lb < p_bs || p_ub > p_bd;
}
```

(b) WP checks
... (c) Loop-directed WP abstraction

... (d) WP consolidation
cwp_p = wpChk(p+k, p+k+2, p_bs, p_bd);
if(...) {
    if(cwp_p) sChk(p+k, p_bs, p_bd, SP);
    a = p[k];
}
if(cwp_p) sChk(p+k+1, p_bs, p_bd, SP);
    b = p[k+1];
...
cwp_a = wpChk(a, a+L+1, a_bs, a_bd);
wp_b = wpChk(b+1, b+L+1, b_bs, b_bd);
// Merging the two WPs in the loop.
wp_loop = cwp_a || wp_b;
// Unswitched loop without checks.
if (!wp_loop) {
    for(i = 1; i <= L; i++) {
        t = a[i-1];
        if(t < ...) {
            t2 = b[i];
            a[i] += t1 + t2;
        }
    }
}
// Unswitched loop with checks.
else {
    for(i = 1; i <= L; i++) {
        sChk(a+i-1, a_bs, a_bd, S);
        t = a[i-1];
        if(t < ...) {
            sChk(b+i, b_bs, b_bd, S);
            t2 = b[i];
            sChk(a+i, a_bs, a_bd, S);
            a[i] += t1 + t2;
        }
    }
}

(e) WP-Driven Loop unswitching

Figure 4.4: A motivating example.

In Figure 4.4(a), there are five memory accesses, four loads (lines 11, 14, 18, and 21) and one store (line 23), with the last three contained in a for loop. With the
unoptimized instrumentation (as obtained by SOFTBOUND), each memory access
triggers a spatial check (highlighted in orange). To avoid cluttering, we do not show
the metadata initialization and propagation, which are irrelevant to our WP-based
optimization.

Value Range Analysis We compute conservatively the value ranges of all point-
ers dereferenced for memory accesses in the program, by using LLVM’s scalar
evolution pass. For the five dereferenced pointers, we have:

\[
\begin{align*}
&p[k] : [p + k \times SP, p + k \times SP] \\
&p[k+1] : [p + (k + 1) \times SP, p + (k + 1) \times SP] \\
&a[i-1] : [a, a + (L - 1) \times S] \\
&b[i] : [b + S, b + L \times S] \\
&a[i] : [a + S, a + L \times S]
\end{align*}
\]

where the two constants, S and SP, are defined at the beginning of the program
in Figure 4.4(a).

Loop-Directed WP Abstraction According to the value ranges computed
above, the WPs for all memory accesses at loads and stores are computed
(weakened if necessary). The WPs for the three memory accesses in the for
loop are found conservatively and hoisted outside the loop to perform a WP
check by calling wpChk in Figure 4.4(b), as shown in Figure 4.4(c). The three
spatial check calls to sChk at a[i-1], b[i] and a[i] that are previously
unconditional (in SOFTBOUND) are now guarded by their WPs, wp_a1, wp_b
and wp_a2, respectively.

Note that wp_a1 is exact since its guarded access a[i-1] will be out-of-bounds
when wp_a1 holds. However, wp_b and wp_a2 are not since their guarded
accesses $b[i]$ and $a[i]$ will never be executed if expression $t < \ldots$ in line 19 always evaluates to false. In general, a WP for an access is constructed so that its falsehood implies the absence of out-of-bounds errors at the access, thereby causing its spatial check to be elided.

The WPs for the other two accesses $p[k]$ and $p[k+1]$ are computed similarly but omitted in Figure 4.4(c).

WP Consolidation In this phase, the WPs for accesses to the same object are considered for consolidation. The code in Figure 4.4(c) is further optimised into the one in Figure 4.4(d), where the two WPs for $p[k]$ and $p[k+1]$ are merged as $cwp_p$ and the two WPs for $a[i-1]$ and $a[i]$ as $cwp_a$. Thus, the number of $wpChk$ calls has dropped from 5 to 3 (lines 2, 10, and 11).

WP-Driven Loop Unswitching This phase generates the final code in Figure 4.4(e). The two WPs in the loop, $cwp_a$ and $wp_b$, are merged as $wp_loop$, enabling the loop to be unswitched. The if branch at lines 16 – 22 is instrumentation-free. The else branch at lines 26 – 35 proceeds as before with the usual spatial checks performed. The key insight for trading code size for performance this way is that the instrumentation-free loop version is often executed more frequently at runtime than its instrumented counterpart in real programs.

4.3.2 The LLVM IR

WPBOUND, as shown in Figure 4.3, works directly on the LLVM-IR, LLVM’s intermediate representation (IR). As illustrated in Figure 4.5, all program variables are partitioned into a set of top-level variables (e.g., $a$, $x$ and $y$) that are not referenced by pointers, and a set of address-taken variables (e.g., $b$ and $c$) that can
be referenced by pointers. In particular, top-level variables are maintained in SSA (Static Single Assignment form) so that each variable use has a unique definition, but address-taken variables are not in SSA.

```c
int **a, *b;
int c, i;
a = &b;
b = &c;
c = 10;
i = c;

x = &c;
a = x;
y = 10;
x = y;
i = *x;
```

(a) C  
(b) LLVM-IR (in pseudocode)

Figure 4.5: The LLVM-IR (in pseudocode) for a C program (where \(x\) and \(y\) are top-level temporaries introduced).

All address-taken variables are kept in memory and can only be accessed (indirectly) via loads (\(q = \star p\) in pseudocode) and stores (\(\star p = q\) in pseudocode), which take only top-level pointer variables as arguments. Furthermore, an address-taken variable can only appear in a statement where its address is taken. All the other variables referred to are top-level.

In the rest of this chapter, we will focus on memory accesses made at the pointer dereferences \(\star p\) via loads \(\cdots = \star p\) and stores \(\star p = \cdots\), where pointers \(p\) are always top-level pointers in the IR. These are the points where the spatial checks are performed as illustrated in Figures 4.2(c) and (d).

Given a pointer \(p\) (top-level or address-taken), its bounds metadata, base (lower bound) and bound (upper bound), are denoted by \(p_{bs}\) and \(p_{bd}\), respectively, as shown in Figure 4.2.
4.3.3 Value Range Analysis

We describe this analysis phase for estimating conservatively the range of values accessed at a pointer dereference, where a spatial check is performed. We conduct our analysis based on LLVM’s scalar evolution pass (Figure 4.3), which calculates closed-form expressions for all top-level scalar integer variables (including top-level pointers) in the way described in [94]. This pass, inspired by the concept of chains of recurrences [4], is capable of handling any value taken by an induction variable at any iteration of its enclosing loops.

A scalar integer expression in the program can be represented as a SCEV (SCalar EVolution expression):

\[ e := c \mid v \mid \mathcal{O} \mid e_1 + e_2 \mid e_1 \times e_2 \mid <e_1, +, e_2>_{\ell} \]

Therefore, a SCEV can be a constant \( c \), a variable \( v \) that cannot be represented by other SCEVs, or a binary operation (involving + and \( \times \) as considered in this work). In addition, when loop induction variables are involved, an add recurrence \(<e_1, +, e_2>_{\ell}\) is used, where \( e_1 \) and \( e_2 \) represent, respectively, the initial value (i.e. the value for the first iteration) and the stride per iteration for the containing loop \( \ell \). For example, in Figure 4.4(a), the SCEV for the pointer \&a[i] contained in the for loop in line 16 is \(<a, +, \text{sizeof(int)}>_{16}\). Finally, the notation \( \mathcal{O} \) is used to represent any value that is neither expressible nor computable in the SCEV framework.

The range of every scalar variable will be expressed in the form of an interval \([e_1, e_2]\). We handle unsigned and signed values differently due to possible integer overflows. According to the C standard, unsigned integer overflow wraps around but signed integer overflow leads to undefined behavior. To avoid potential over-
Chapter 4. Accelerating Detection of Spatial Errors

flows, we consider conservatively the range of an unsigned integer variable as $[\mathcal{O}, \mathcal{O}]$.

For operations on signed integers, we assume that overflow never occurs. This assumption is common in compiler optimizations. For example, the following function (with $x$ being a signed int):

```c
bool foo(int x) { return x + 1 < x; }
```

is optimised by LLVM, GCC and ICC to return false.

Figure 4.6: Range analysis rules.

The rules used for computing the value ranges of signed integer and pointer variables are given in Figure 4.6. [Termi] suggests that both the lower and upper bounds of a SCEV, which is $c$, $v$ or $\mathcal{O}$, are the SCEV itself. [Add] asserts that the lower (upper) bound of an addition SCEV $e_1 + e_2$ is simply the lower (upper) bounds of its two operands added together. When it comes to a multiplication SCEV, the usual min and max functions are called for, as indicated in [Mul]. If $\min(V)$ and $\max(V)$ cannot be solved statically at compile time, then $[\mathcal{O}, \mathcal{O}]$ is
assumed. For example, $[i, i + 10] \times [2, 2] \downarrow [2i, 2i + 20]$ but $[i, 10] \times [j, 10] \downarrow [\mathcal{O}, \mathcal{O}]$, where $i$ and $j$ are scalar variables. In the latter case, the compiler cannot statically resolve $\min(V)$ and $\max(V)$, where $V = \{10i, 10j, ij, 100\}$.

For an add recurrence, the LLVM scalar evolution pass computes the trip count of its containing loop $\ell$, which is also represented as a SCEV $tc(\ell)$. A trip count can be $\mathcal{O}$ since it may neither be expressible nor computable in the SCEV formulation. In the case of a loop with multiple exits, the worst-case trip count is picked. Here, we assume that a trip count is always positive. However, this will not affect the correctness of our overall approach, since the possibly incorrect range information is never used inside a non-executed loop.

In addition to some simple scenarios demonstrated in our motivating example, our value range analysis is capable of handling more complex ones, as long as LLVM’s scalar evolution is. Consider the following double loop:

```c
for (int i = 0; i < N; ++i) // L1
  for (int j = 0; j <= i; ++j) // L2
    a[2*i+j] = ...; // a declared as int*
```

The SCEV of $\&a[2*i+j]$, i.e., $a + 2*i + j$ is given as $\langle \langle a, +, 2 \times \text{sizeof(int)} \rangle_{L_1}, +, \text{sizeof(int)} \rangle_{L_2}$ by scalar evolution, and $tc(L_1)$ and $tc(L_2)$ are $N$ and $<0, +, 1>_{L_1} + 1$, (i.e., $i+1$), respectively. The value range of $\&a[2*i+j]$ is then deducted via the rules in Figure 4.6 as:

$$[a, a + 3 \times (N - 1) \times \text{sizeof(int)}]$$
4.3.4 WP-based Instrumentation

We describe how WPBOUND generates the instrumentation code for a program during its three transformation phases, based on the results of value range analysis. We only discuss how bounds checking operations are inserted since WPBOUND handles metadata initialization and propagation exactly as in SOFTBOUND, as illustrated in Figure 4.2.

Transformation I: Loop-Directed WP Abstraction

This phase computes the WPs for all dereferenced pointers and inserts guarded or unguarded spatial checks for them. As shown in our motivating example, we do so by reasoning about the WP for a pointer \( p \) at a load \( \cdots = \ast p \) or a store \( \ast p = \cdots \). Based on the results of value range analysis, we estimate the WP for \( p \) according to its Memory Access Region (MAR), denoted \([p_{lb}^{mar}, p_{ub}^{mar}]\). Let the value range of \( p \) be \([p_l, p_u]\). There are two cases:

1. \( p_l \neq \mathcal{O} \land p_u \neq \mathcal{O} \): \([p_{lb}^{mar}, p_{ub}^{mar}] = [p_l, p_u + \text{sizeof}(\ast p)]\). As a result, its WP is estimated to be:

\[
p_{lb}^{mar} < p_{bs} \lor p_{ub}^{mar} > p_{bd}
\]

where \( p_{bs} \) and \( p_{bd} \) are the base and bound of \( p \) (Section 4.3.2). The result of evaluating this WP, called a WP check, can be obtained by a call to \( \text{wpChk}(p_{lb}^{mar}, p_{ub}^{mar}, p_{bs}, p_{bd}) \) in Figure 4.4(b).

2. \( p_l = \mathcal{O} \lor p_u = \mathcal{O} \): The MAR of \( p \) is \([p_{lb}^{mar}, p_{ub}^{mar}] = [\mathcal{O}, \mathcal{O}]\) conservatively. The WP is set as true.

In general, the WP thus constructed for \( p \) is not the weakest one, i.e., the one ensuring that if it holds during program execution, then some accesses via \( \ast p \) must be out-of-bounds. There are two reasons for being conservative. First, value
range analysis is imprecise. Second, all branch conditions (e.g., the one in line 19 in Figure 4.4) affecting the execution of \(*p\) are ignored during this analysis, as explained in Section 4.3.1.

However, by construction, the falsehood of the WP for \(p\) always implies the absence of out-of-bounds errors at \(*p\), in which case the spatial check at \(*p\) can be elided. However, the converse may not hold, implying that some bounds checking operations performed when the WP holds are redundant.

After the WPs for all dereferenced pointers in a program are available, \textsc{Instrument}(\(F\)) in Algorithm 2 is called for each function \(F\) in the program to guard the spatial check at each pointer dereference \(*p\) by its WP when its MAR is neither \([O, O]\) (in which case, its WP is true) nor loop-variant. In this case (lines 4 – 6), the guard for \(p\), which is loop-invariant at point \(s\), is hoisted to the point identified by \textsc{PositioningWP}(), where it is evaluated. The spatial check at the pointer dereference \(*p\) becomes conditional on the guard. Otherwise (line 7), the spatial check at the dereference \(*p\) is unconditional as is the case in \textsc{SoftBound}.

Note that an access \(*p\) may appear in a set of nested loops. \textsc{PositioningWP} returns the point just before the loop at the highest depth for which the WP for \(p\) is loop-invariant and \(p\) (representing the point where \(*p\) occurs) otherwise.

Let us return to Figure 4.4(c). The MAR of \(b[i]\) in line 10 is \([b + SZ, b + (L + 1) \times SZ]\), whose lower and upper bounds are invariants of the \textsc{for} loop in line 5. With the WP check, \(wp\_b\), evaluated in line 4, the spatial check for \(b[i]\) inserted in line 9 is performed only when \(wp\_b\) is true.

Compared to \textsc{SoftBound} that produces the unguarded instrumentation code as explained in Section 4.2, our WP-based instrumentation may increase code size slightly. However, many WPs are expected to be true in real programs. Instead of the five instructions, \texttt{cmp}, \texttt{br}, \texttt{lea}, \texttt{cmp} and \texttt{br}, required for performing a spatial
Algorithm 2 Loop-Directed WP Abstraction

Procedure INSTRUMENT($F$)
begin
1  foreach pointer dereference $*p$ in function $F$ do
2    Let $SIZE$ be $sizeof(*p)$;
3    $s \leftarrow$ POSITIONINGWP($p$);
4    if $[p_{lb}^{mar}, p_{ub}^{mar}] \neq [O, O] \land s \neq p$ then
5      Insert a $wpChk$ call for $*p$ at point $s$:
6      $wp_p = wpChk(p_{lb}^{mar}, p_{ub}^{mar}, p_{bs}, p_{bd})$;
7      Insert a guarded spatial check before $*p$:
8      if ($wp_p$) $sChk(p, p_{bs}, p_{bd}, SIZE)$;
9      else
10     Insert an unguarded spatial check before $*p$:
11     $sChk(p, p_{bs}, p_{bd}, SIZE)$;

Procedure POSITIONINGWP($p$)
begin
8  $s \leftarrow p$; // denoting the point where $*p$ is
9  while $s$ is inside a loop do
10  Let $\ell$ be the innermost loop containing $s$;
11  if $p_{lb}^{mar}$ and $p_{ub}^{mar}$ are invariants in $\ell$ then
12  $s \leftarrow$ the point just before $\ell$;
13  else break;
14  return $s$;

check, $sChk$, two instructions, $cmp$ and $br$, are usually executed to test its guard only.

Transformation II: WP Consolidation

This phase conducts an intraprocedural analysis to combine the WPs corresponding to a set of memory accesses to the same object (e.g., the same array) into a single one to be shared (e.g., $cwp_p$ and $cwp_a$ in Figure 4.4(d)). If a pointer dereference is not in a loop, its spatial check is not guarded according to Algorithm 2 (since $s = p$ in line 3). By combining its WP with others, we will also make such a check
guarded as well (e.g., $\text{cwp}_p$ in Figure 4.4(d)).

Algorithm 3 is simple. Given a function $F$, where $W$ initially contains all pointers dereferenced at loads and stores in $F$ (line 1), we start with $G = \{p\}$ (line 4). We then add iteratively all other pointers $q_1, \ldots, q_n$ in $F$ (lines 6 – 15) to $G = \{p, q_1, \ldots, q_n\}$, so that the following properties hold:

**Prop. 1** All these pointers point to the same object. If $q$ selected in line 6 does not point to the same object as $p$, $p_{lb}^{\prime}$ or $p_{ub}^{\prime}$ will be $O$, causing $s_p^i = \epsilon$ (due to line 22). In this case, $q$ will not be added to $G$ (line 11).

**Prop. 2** The WPs for these pointers are invariants with respect to point $s_p$ found at the end of the foreach loop in line 6 (due to lines 23 – 27). As all variables in $V$ (line 23) are in SSA, the definition of $v$ in line 25 is unique.

When $|G| > 1$ (line 16), we can combine the WPs in $G$ into a single one, $\text{cwp}_G$ (line 17), where $[p_{lb}^{\text{mar}}, p_{ub}^{\text{mar}}]$ is constructed to be the union of the MARs of all the pointers in $G$. Note that $\text{wpChk}$ is called only once since $\forall q \in G : q_{bs} = p_{bs} \land q_{bd} = p_{bd}$ by construction. In lines 18 – 20, the spatial checks for all pointers in $G$ are modified to use $\text{cwp}_G$ instead.

Consider Figure 4.4(d) again. The MARs for $a[i-1]$ in line 14 and $a[i]$ in line 19 are $[a, a + L \times SZ)$ and $[a + SZ, a + (L + 1) \times SZ)$, respectively. The consolidated MAR is $[a, a + (L + 1) \times SZ)$, yielding a WP $\text{cwp}_a$ weaker than the WPs, $\text{wp}_a1$ and $\text{wp}_a2$, for $a[i-1]$ and $a[i]$, respectively. The WP check $\text{cwp}_a$ is inserted in line 10, which dominates $a[i-1]$ and $a[i]$ in the CFG. The spatial checks for $a[i-1]$ and $a[i]$ are now guarded by $\text{cwp}_a$.

**Transformation III: WP-Driven Loop Unswitching**

In this last intraprocedural phase, we apply loop unswitching, a standard loop transformation, to a loop, as illustrated in Figure 4.4(e), to unswitch some guarded
Algorithm 3 WP Consolidation

Procedure CONSOLIDATEWP(F)

begin

\[ W \leftarrow \text{set of pointers dereferenced in function } F; \]

\[ \text{while } W \neq \emptyset \text{ do} \]

\[ p \leftarrow \text{a pointer from } W; \]

\[ G \leftarrow \{ p \}; \]

\[ s_p \leftarrow \text{POSITIONINGWP}(p); \]

\[ \text{foreach } q \in W \text{ such that } q \neq p \text{ do} \]

\[ s_q \leftarrow \text{POSITIONINGWP}(q); \]

\[ p_{lb}^m \leftarrow \min\{p_{lb}^m, q_{lb}^m\}; \]

\[ p_{ub}^m \leftarrow \max\{p_{ub}^m, q_{ub}^m\}; \]

\[ s_p' \leftarrow \text{DOMINATOR}(F, s_p, s_q, p_{lb}^m, p_{ub}^m); \]

\[ \text{if } s_p' \neq \epsilon \text{ then} \]

\[ G \leftarrow G \cup \{ q \}; \]

\[ p_{lb}^m \leftarrow p_{lb}'; \]

\[ p_{ub}^m \leftarrow p_{ub}'; \]

\[ s_p \leftarrow s_p'; \]

\[ \text{if } G \neq \{ p \} \text{ then} \]

\[ \text{Insert a wpChk call for } *p \text{ at point } s_p; \]

\[ cwp_G = \text{wpChk}(p_{lb}^m, p_{ub}^m, q_{bs}, q_{bd}); \]

\[ \text{foreach } q \in G \text{ do} \]

\[ \text{Let } SIZE \text{ be sizeof}(q); \]

\[ \text{Replace the spatial check for } *q \text{ by:} \]

\[ \text{if } (cwp_G) \text{ sChk}(q, q_{bs}, q_{bd}, SIZE); \]

\[ W \leftarrow W - G; \]

Procedure DOMINATOR(F, s_1, s_2, p_l, p_u)

begin

\[ \text{if } p_l = \emptyset \vee p_u = \emptyset \text{ then } \text{return } \epsilon; \]

\[ V \leftarrow \{ v \mid \text{variable } v \text{ occurs in SCEV } p_l \text{ or SCEV } p_u \}; \]

\[ S \leftarrow \text{set of (program) points in the CFG of } F; \]

\[ \text{if } \exists s \in S : \{ s \text{ dominates } s_1 \text{ and } s_2 \text{ in } F\text{'s CFG} \} \land \]

\[ (\forall v \in V : \text{the def of } v \text{ dominates } s \text{ in } F\text{'s CFG}) \text{ then} \]

\[ \text{return } s; \]

\[ \text{else} \]

\[ \text{return } \epsilon; \]
spatial checks, so that its guards are hoisted outside the loop, resulting in their repeated tests inside the loop being effectively removed in some versions of the loop. However, unswitching all branches in a loop may lead to code growth exponential in its number of branches.

To avoid code explosion, we apply Algorithm 4 to a function $F$ to process its loops inside out. For a loop $\ell$ (line 2), we first partition a set $S$ of its guarding WPs selected in line 3 into a few groups (discussed below in more detail) (line 5). We then insert a disjunction $wp_\pi$ built from the WPs in each group $\pi$ just before $\ell$ (line 7). As $wp_\pi$ is weaker than each constituent $wp$, we can replace each $wp$ by $wp_\pi$ at the expense of more spatial checks (lines 8 – 9). Finally, we unswitch loop $\ell$ so that each spatial check guarded by $wp_\pi$ is either performed unconditionally (in its true version) or removed (in its false version). As these “unswitched” checks will not be considered again (line 3), our algorithm will eventually terminate.

Algorithm 4 WP-Driven Loop Unswitching

Procedure LoopUnswitching($F$)

begin
1 $\mathcal{L}$ ← a loop nest forest obtained in function $F$;
2 foreach loop $\ell$ in reverse topological order in $\mathcal{L}$ do
3 \hspace{1em} $S$ ← \{$wp$ | (1) “if (wp) sChk(...)” is inside $\ell$ \wedge
4 \hspace{2em} (2) $wp$ is an invariant in $\ell$ \wedge
5 \hspace{3em} (3) ($\exists \ell' \in \mathcal{L} : \ell'$ contains $\ell$ \wedge $wp$ is an invariant in $\ell'$)$\}$;
6 \hspace{1em} if $S = \emptyset$ then continue;
7 $\Pi$ ← a partition of $S$ into groups;
8 foreach group $\pi \in \Pi$ do
9 \hspace{1em} Insert $wp_\pi \leftarrow \bigvee_{wp \in \pi} wp$ just outside $\ell$;
10 \hspace{1em} foreach $wp \in \pi$ do
11 \hspace{2em} Replace each $wp$ inside $\ell$ by $wp_\pi$;
12 \hspace{1em} Unswitch $\ell$ for every $wp_\pi$, where $\pi \in \Pi$;

Let us discuss the three conditions used in determining a set $S$ of guarding WPs
to unswitch in line 3. Condition (1) instructs us to consider only guarded special checks. Condition (2) avoids any guarding WP that is loop-variant since it may be introduced by Algorithm 3. Condition (3) allows us to exploit a sweet-spot to make a tradeoff between code size and performance for real code. Without (3), $S$ tends to be larger, leading to weaker $wp_r$’s than otherwise. As a result, we tend to generate fewer loop versions, by trading performance for code size. With (3), the opposite tradeoff is made.

In line 5, there can be a number of ways to partition $S$. In general, a fine-grained partitioning eliminates more redundant bounds checks than a coarse-grained partitioning, but results in more code versions representing different combinations of instrumented and un-instrumented memory accesses. Note that the space complexity (i.e., code expansion) of loop unswitching is exponential to $|\Pi|$, i.e., the number of partitions.

To keep code sizes manageable in our implementation of this algorithm, we have adopted a simple partitioning strategy by setting $\Pi = \{S\}$. Together with Conditions (1) – (3) in line 3, this partitioning strategy is effective in practice.

Let us apply our algorithm to Figure 4.4(d) to unswitch the for loop, which contains two WP guards, $cwp_a$ and $wp_b$. Replacing them with a weaker one, $wp_{\text{loop}} = cwp_a \lor wp_b$ and then unswitching the loop yields the final code in Figure 4.4(e). The instrumentation-free version appears in lines 16 – 22 and the instrumented one in lines 26 – 35.

4.4 Evaluation

The goal of this evaluation is to demonstrate that our WP-based tool, WPBOUND, can significantly reduce the runtime overhead of SOFTBOUND, a state-of-the-art
instrumentation tool for enforcing spatial memory safety of C programs.

### 4.4.1 Implementation Considerations

Based on the open-source code of SOFTBOUND, we have implemented WPBOUND also in LLVM (version 3.3). In both cases, the bounds metadata are maintained in a separate shadow space. Like SOFTBOUND, WPBOUND handles a number of issues identically as follows:

- Array indexing (also for multiple-dimensional arrays) is handled equivalently as pointer arithmetic.

- The metadata for global pointers are initialized, by using the same hooks that C++ uses for constructing global objects.

- For external function uses in un-instrumented libraries, we resort to SOFTBOUND’s library function wrappers (Figure 4.7), which enforce the spatial safety and summarize the side effects on the metadata.

- For a function pointer, its bound equals to its base, describing a zero-sized object that is not used by data objects. This prevents data pointers or non-pointer data from being interpreted as function pointers.

- For pointer type conversions via either explicit casts or implicit unions, the bounds information simply propagates across due to the disjoint metadata space used.

- Finally, we do not yet enforce the spatial safety for variable argument functions. A possible solution is to introduce an extra argument describing the number of arguments passed (in bytes), so that the `va_start` and `va_arg` could check if there are too many arguments are decoded.
4.4.2 Experimental Setup

All experiments are conducted on a machine equipped with a 3.00GHz quad-core Intel Core2 Extreme X9650 CPU and 8GB DDR2 RAM, running on a 64-bit Ubuntu 10.10. The SoftBound tool is taken from the SoftBoundCETS open-source project (version 1.3) [61, 62], configurated to enforce spatial memory safety only.

Table 4.1 lists a set of 12 SPEC benchmarks used. These benchmarks are often used in the literature [1, 35, 60, 61, 76]. We have selected eight from the 12 C benchmarks in the SPEC2006 suite, by excluding gcc and perlbench since both cannot be processed correctly under SoftBound (as described in its README) and gobmk and sjeng since these two game applications are not loop-oriented. In addition to SPEC2006, we have included four loop-oriented SPEC2000 benchmarks, ammp, art, gzip and twolf, in our evaluation.

4.4.3 Methodology

Figure 4.7 shows the compilation workflow for both SoftBound and WPBound in our experiments. All source files of a program are compiled under the “-O2” flag and then merged into one bitcode file using llvm-link. The instrumentation code is inserted into the merged bitcode file by a SoftBound or WPBound pass. Then the bitcode file with instrumentation code is linked to the SoftBound runtime library to generate binary code, with the link-time optimization flag “-O2” used to further optimize the instrumentation code inserted.

To analyze the runtime overheads introduced by both tools, the native (instrumentation-free) code is also generated under the “-O2” together with link-time optimization.
4.4.4 Instrumentation Results

Let us first discuss the instrumentation results of the 12 benchmarks according to the statistics given in Table 4.2.

In Column 2, we see that SOFTBOUND inserts an average of 5035 spatial checks for each benchmark. Note that the number of spatial checks inserted is always smaller than the number of loads and stores added together. This is because SOFTBOUND has eliminated some unnecessary spatial checks by applying some simple optimizations including its dominator-based redundant check elimination [61]. This set of optimizations is also performed by WPBOUND as well.

In Columns 3 – 7, we can observe some results collected for WPBOUND. According to Column 3, there are an average of 719 wpChk calls inserted in each benchmark by Algorithm 2 (for WP-based instrumentation), causing \( \sim 1/7 \) of the spatial checks inserted by SOFTBOUND to be guarded. According to Column 4,
Algorithm 3 (for WP consolidation) has made an average of 2073 unconditional checks guarded (a reduction of 41%) for each benchmark. According to Column 5, Algorithm 4 (for loop unswitching) has succeeded in merging an average of 192 WPs at loop entries for each benchmark. Overall, the average number of the WPs combined to yield one disjunctive WP is 5.6 (Column 6), peaking at 235 constituent WPs in one disjunctive WP in the `Mode_Decision_for_4x4IntraBlocks` function in `h264ref` (Column 7).

Finally, as compared in Figure 4.8, WPBOUND results in slightly larger code sizes than SOFTBOUND due to (1) the `wpChk` calls introduced, (2) the guards added to some spatial checks, and (3) code duplication caused by loop unswitching. Compared to un-instrumented native code, the geometric means of code size increases for SOFTBOUND and WPBOUND are 1.72X and 2.12X, respectively. This implies

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Functions</th>
<th>#Loads</th>
<th>#Stores</th>
<th>#Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>180</td>
<td>3,705</td>
<td>1,187</td>
<td>650</td>
</tr>
<tr>
<td>art</td>
<td>27</td>
<td>471</td>
<td>182</td>
<td>158</td>
</tr>
<tr>
<td>gzip</td>
<td>72</td>
<td>936</td>
<td>711</td>
<td>257</td>
</tr>
<tr>
<td>twolf</td>
<td>188</td>
<td>9,781</td>
<td>3,304</td>
<td>1,253</td>
</tr>
<tr>
<td>bzip2</td>
<td>68</td>
<td>2,570</td>
<td>1,680</td>
<td>545</td>
</tr>
<tr>
<td>h264ref</td>
<td>517</td>
<td>20,984</td>
<td>8,277</td>
<td>2,698</td>
</tr>
<tr>
<td>hmmer</td>
<td>472</td>
<td>8,345</td>
<td>3,608</td>
<td>1,667</td>
</tr>
<tr>
<td>lbm</td>
<td>18</td>
<td>244</td>
<td>114</td>
<td>32</td>
</tr>
<tr>
<td>libquantum</td>
<td>96</td>
<td>604</td>
<td>317</td>
<td>144</td>
</tr>
<tr>
<td>mcf</td>
<td>26</td>
<td>347</td>
<td>224</td>
<td>76</td>
</tr>
<tr>
<td>milc</td>
<td>236</td>
<td>3,443</td>
<td>1,094</td>
<td>544</td>
</tr>
<tr>
<td>sphinx3</td>
<td>320</td>
<td>4,628</td>
<td>1,359</td>
<td>1,240</td>
</tr>
<tr>
<td>ArithMean</td>
<td>185</td>
<td>4,672</td>
<td>1,838</td>
<td>772</td>
</tr>
</tbody>
</table>

Table 4.1: Benchmark statistics.
that WPBOUND has made an instrumented program about 23% larger than SOFTBOUND on average. In general, the code explosion problem is well contained due to the partitioning heuristics used in our WP-based loop unswitching as discussed in Section 4.3.4.
4.4.5 Performance Results

To understand the effects of our WP-based approach on performance, we compare WPBOUND and SOFTBOUND in terms of their overheads and the number of checks performed.

(I) Runtime Overheads

Figure 4.9 compares WPBOUND and SOFTBOUND in terms of their runtime slowdowns over the native code (as the un-instrumented baseline). The average overhead of a tool is measured as the geometric mean of overhead of all benchmarks analyzed by the tool.

SOFTBOUND exhibits an average overhead of 71%, reaching 180% at h264ref. In the case of our WP-based instrumentation, WPBOUND has reduced SOFTBOUND’s average overhead from 71% to 45%, with significant reductions achieved.
Figure 4.9: Execution time (normalized with respect to native code).
at *hmmer* (73%), *libquantum* (91%) and *milc* (57%). For *lbm*, which is the best case for both tools, *SOFTBOUND* and *WPBOUND* suffer from only 3.7% and 0.9% overheads, respectively. In this benchmark, the pointer load and store operations that are costly for in-memory metadata propagations (as shown in in Figure 4.2(e)) are relatively scarce. In addition, *SOFTBOUND*’s simple dominator-based redundant check elimination identifies 60% of the checks as unnecessary.

(II) Dynamic Check Count Reduction

Figure 4.10 shows the ratios of the dynamic number of checks, i.e., calls to *wpChk* and *sChk* executed under *WPBOUND* over the dynamic number of checks, i.e., calls to *sChk* executed under *SOFTBOUND* (in percentage terms). On average, *WPBOUND* performs only 36.0% of *SOFTBOUND*’s checks, comprising 5.2% *wpChk* calls and 30.8% *sChk* calls. For every benchmark considered, the number of checks performed by *WPBOUND* is always lower than that performed by *SOFTBOUND*. This confirms that the WPs constructed by *WPBOUND* for real code typically evaluate to true, causing their guarded checks to be avoided.

(III) Correlation

By comparing Figures 4.9 and 4.10, we can observe that *WPBOUND* is usually effective in reducing bounds checking overheads in programs where it is also effective in reducing the dynamic number of checks performed by *SOFTBOUND*. This has been the case for benchmarks such as *hmmer*, *lbm*, *libquantum* and *milc*. As for *bzip2*, *WPBOUND* still preserves 85% of *SOFTBOUND*’s checks, thereby reducing its overhead from 78% to 74% only.

We also observe that a certain percentage reduction in the dynamic number of checks achieved by *WPBOUND* does not translate into execution time benefits
Chapter 4. Accelerating Detection of Spatial Errors

4.5 Related Work

In addition to the pointer-based approaches described in Section 4.2, we now recall the guard zone-based and object-based approaches for enforcing spatial safety as previously introduced in Section 2.2. We also discuss some other related work on bounds check elimination and static analysis.
4.5.1 Guard Zone-based Spatial Safety

Guard zone-based approaches \([35, 36, 66, 76, 107]\) enforce spatial safety by placing a guard zone of invalid memory between memory objects. Continuous overflows caused by walking across a memory object’s boundary in small strides will hit a guard zone, resulting in an out-of-bounds error. In the case of overflows with a large stride that jumps over a guard zone and falls into another memory object, an out-of-bounds error will be missed. As a result, these approaches provide neither source compatibility nor complete spatial safety.

4.5.2 Object-based Spatial Safety

In object-based approaches \([1, 15, 17, 21, 41, 75]\), the bounds information is maintained per object (rather than per-pointer as in pointer-based approaches). In addition, the bounds information of an object is associated with the location of the object in memory. As a result, all pointers to an object share the same bounds information. On every pointer-related operation, a spatial check is performed to ensure that the memory access is within the bounds of the same object.

Compared to pointer-based approaches, object-based approaches usually have better compatibility with un-instrumented libraries. The metadata associated with heap objects are properly updated by interpreting \texttt{malloc} and \texttt{free} function calls, even if the objects are allocated or de-allocated by un-instrumented code. Unlike pointer-based approaches, however, object-based approaches do not provide complete spatial safety, since sub-object overflows (e.g., overflows of accesses to arrays inside structs) are missed.

Note that our WP-based optimization can be applied to guard zone- and object-based approaches, although we have demonstrated its effectiveness in the context of a pointer-based approach, which has recently been embraced by Intel in a recently
released commercial software product [27].

4.5.3 Bounds Check Elimination

Bounds check elimination, which reduces the runtime overhead incurred in checking out-of-bounds array accesses for Java, has been extensively studied in the literature [5, 24, 25, 57, 68, 72, 96, 97]. One common approach relies on solving a set of constraints formulated based on the program code [5, 25, 68, 72]. Another is to speculatively assume that some checks are unnecessary and generate check-free specialized code, with execution reverted to unoptimized code when the assumption fails [24, 96, 97].

Loops in the program are also a target for bounds check elimination [57]. Some simple patterns can be identified, where unnecessary bound checks can be safely removed.

SOFTBOUND [61] applies simple compile-time optimizations including a dominator-based redundant check elimination to eliminate unnecessary checks dominated by other checks.

Our WP-based optimization complements prior work by making certain spatial checks guarded so that a large number of spatial checks are avoided conditionally.

4.5.4 Static Analysis

A significant body of work exists on statically detecting and diagnosing buffer overflows [3, 9, 18, 19, 26, 29, 31, 45, 47, 55, 74, 100]. Due to its approximation nature, static analysis alone finds it rather difficult to maintain both precision and efficiency, and generally has either false positives or false negatives. However, its precision can be improved by using modern pointer analysis [34, 42, 46, 51, 78, 79, 80, 83, 88, 91, 95, 101, 106, 109] and value-flow analysis [48, 49, 89, 90, 92]
techniques. In the work proposed in Chapter 3, static value-flow analysis has been combined with dynamic analysis to reduce instrumentation overheads in detecting uninitialised variables. So existing static analysis techniques can be exploited to compute WPs more precisely for our WP-based instrumentation.

In addition, the efficiency of static analysis techniques can be improved if they are tailored to specific clients. Dillig et al. [18] have recently proposed a static analysis to compute the preconditions for dictating spatial memory safety conservatively. Rather than analysing the entire program, their static analysis works in a demand-driven manner, where the programmer first specifies a code snippet as a query and then the proposed static analysis infers a guard to ensure spatial memory safety for the code snippet. Such analysis uses logical abduction and is thus capable of computing the weakest and simplest guards. In contrast, our work is based on the symbolic analysis of LLVM’s scalar evolution and thus more lightweight as an optimization for whole-program spatial-error detection.

4.6 Chapter Summary

In this chapter, we introduce a new WP-based compile-time optimization to enforce spatial memory safety for C. Our optimization complements existing bounds checking optimizations and can be applied to any spatial-error detection approaches (in software or hardware). Implemented on top of SOFTBOUND, a state-of-the-art tool for detecting spatial errors, our WP-based instrumentation tool, WPBOUND, provides compatible and comprehensive spatial safety (by maintaining disjoint per-pointer metadata as in SOFTBOUND) and supports separate compilation (since all its four phases are intraprocedural). For a set of 12 SPEC C benchmarks evaluated, WPBOUND, can substantially reduce the runtime overheads incurred by
SOFTBOUND with small code size increases. The work proposed in this chapter was previously published in [104].
Chapter 5

Conclusions

This chapter firstly summarizes the thesis in Section 5.1. Future work, including possible extensions to this thesis and some potential future research directions, is then discussed in Section 5.2.

5.1 Thesis Summary

Memory errors in C programs have been one of the major threats to program safety, system security, and software reliability. Runtime detection is a practical solution to tackle memory errors. Although the memory error detection techniques have been studied for a long time, the instrumentation overheads incurred in existing detection tools are still significant. How to make the detection more efficient is a crucial problem, especially when performance is a major concern.

In this thesis, we addressed the problem by reducing unnecessary instrumentation guided by static analysis. Our focus is on two types of memory errors: (1) uses of undefined values, and (2) spatial errors. They are of different program features and require different sets of techniques for detection. Undefined values are caused by uninitialized variables. They can cause system crashes when used
and security vulnerabilities when exploited. With source rather than binary instrument-
tation, dynamic analysis tools such as MSAN can detect uninitialized memory
uses at significantly reduced overheads but are still costly. Spatial errors (e.g.,
buffer overflows) continue to be one of the dominant threats to software reliability
and security in C/C++ programs. Presently, the software industry typically en-
forces spatial memory safety by instrumentation. Due to high overheads incurred
in bounds checking at runtime, many program inputs cannot be exercised, caus-
ing some input-specific undefined value uses and spatial errors to go undetected in
today’s commercial software.

For efficient detection of undefined value uses, we introduced a static value-flow
analysis, called Usher, to guide and speed up the dynamic analysis performed by
such tools. Usher infers the definedness of values using a value-flow graph that
captures def-use chains for both top-level and address-taken variables interproce-
durally and removes unnecessary instrumentation by solving a graph reachability
problem. Usher works well with any pointer analysis (done a priori) and facili-
tates advanced instrumentation-reducing optimizations, two being demonstrated
in this thesis. Implemented in LLVM and evaluated using all the 15 SPEC2000 C
programs, Usher can reduce the slowdown of MSAN from 212% – 302% to 123%
– 140% for a number of configurations tested.

For efficient spatial error detection, we introduced a new compile-time opti-
mization for reducing bounds checking overheads based on the notion of weakest
precondition. The basic idea is to guard a bounds check at a pointer dereference
inside a loop, where the WP-based guard is hoisted outside the loop, so that its
falsehood implies the absence of out-of-bounds errors at the dereference, thereby
avoiding the corresponding bounds check inside the loop. This WP-based optimiza-
tion is applicable to any spatial-error detection approach (in software or hardware
or both). To evaluate the effectiveness of our optimization, we take SOFTBOUND, a compile-time tool with an open-source implementation in LLVM, as our baseline. SOFTBOUND adopts a pointer-based checking approach with disjoint metadata, making it a state-of-the-art tool in providing compatible and complete spatial safety for C. Our new tool, called WPBOUND, is a refined version of SOFTBOUND, also implemented in LLVM, by incorporating our WP-based optimization. For a set of 12 SPEC C benchmarks evaluated, WPBOUND reduces the average runtime overhead of SOFTBOUND from 71% to 45% (a reduction of 37%), with small code size increases.

We conclude that it is possible to enhance the performance of dynamic memory error detection with the assistance of static analysis. Static value-flow analysis and conservative weakest precondition approximation can be used to tackle undefined value uses and spatial errors, respectively. As a result, the runtime overheads of detecting these errors can be significantly reduced, as demonstrated by USHER and WPBOUND.

5.2 Future Work

Although the solutions described in this thesis have been demonstrated to be effective in reducing performance overheads incurred by instrumentation, there are still a number of interesting aspects that can be potentially extended to make them more powerful.

5.2.1 Detecting Other Memory Errors

The proposed techniques can be applied to detect other types of memory errors, though we have only studied two cases – (1) use of undefined values and (2) spatial
error detection – in the context of this thesis. Value-flow analysis is used to track the definedness of values in memory objects for efficient detection of undefined value uses, so the performance of temporal error and memory leak detection can be enhanced similarly, with corresponding extensions.

For efficient temporal error detection, we could introduce a new state $D$ for shadow values to Usher, where $D$ indicates that the variable has been de-allocated. To be more specific, every shadow value can be $T$, $F$, or $D$. When a memory object is de-allocated, Usher marks its shadow bits to be $D$. For every pointer dereferenced by a memory access or passed to a `free()` function call, Usher checks whether the variable pointed-to by the pointer has a $D$ shadow value. If so, a use-after-free or double free error is captured. By using similar value-flow analysis techniques, the instrumentation overhead for temporal error detection can be reduced.

To facilitate efficient runtime detection of memory leaks, we are able to use the results of our previously proposed value-flow analysis \cite{89, 90} to filter out some unnecessary instrumentation.

### 5.2.2 Extensions for Usher and WPBOUND

Usher and WPBOUND have already been demonstrated to be efficient compared to previous tools. There are, however, some possible solutions for further speedups.

To further enhance the performance of Usher, we may:

- leverage more precise pointer analysis developed by our group \cite{51, 78, 79, 88, 91, 92, 106, 109} and others \cite{34, 42, 46, 48, 49, 80, 83, 95, 101} to increase the precision of value-flow analysis;

- use adaptive context-sensitivity to resolve graph reachability, rather than
universally using 1-callsite context-sensitivity;

- perform some shape analysis to infer the definedness of heap objects in a more precise manner;

- employ parallel pointer analysis techniques developed by us [85, 86, 87] and others [20, 53, 54, 58, 63, 71] to speedup static analysis.

To make WPBOUND more efficient, we may:

- use more powerful and dedicated value range analysis to identify potential integer overflows;

- apply polyhedral techniques to handle some loops whose bounds cannot be trivially identified by the current implementation of WPBOUND;

- integrate some heavyweight analysis to approximate weakest preconditions more precisely (e.g., SMT-based techniques to analyze the constraints of conditions);

- perform interprocedural analysis to eliminate checks for memory accesses inside the callees that take residence in a loop of their callers.

### 5.2.3 Accelerating Error Detection for Other Languages

Apart from C programs, it is also possible to apply the guided instrumentation techniques for error detection in software written in other languages. In Java, for example, the instrumentation overhead for data race detection [23, 81, 98, 99] is significant. We may employ static program analysis techniques for Java, such as [50, 83, 102], to guide the instrumentation for better performance.
5.2.4 Static Analysis Guided by Dynamic Information

In this thesis, we make use of the static analysis results to guide dynamic analysis by reducing unnecessary program instrumentation. On the other hand, static and dynamic analysis can be possibly integrated in an inverse manner, where runtime information is used to guide static analysis.

Since the effectiveness of performance increase depends on the precision of static analysis, one possible solution is to use profiling tools to identify some hot regions of a program. Then, we may apply some heavyweight static analysis, which is not scalable for the whole program, for these hot regions only. The precision of static analysis results for the focused code can then be improved, and the corresponding instrumentation for these hot regions is likely to become more efficient.
Bibliography


http://software.intel.com/sites/products/


flow aware constraint analyzer for bounds check elimination. In SAS, pages


[70] H. Patil and C. N. Fischer. Low-cost, concurrent checking of pointer and

[71] S. Putta and R. Nasre. Parallel replication-based points-to analysis. In CC,

[72] F. Qian, L. J. Hendren, and C. Verbrugge. A comprehensive approach to


[74] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array

[75] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In

[76] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer:

[77] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors

[78] L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with in-
cremental CFL-reachability summarisation: preliminary experience. In ASE,


