PRACTICAL LOW-OVERHEAD ENFORCEMENT OF MEMORY SAFETY FOR C PROGRAMS

Santosh Ganapati Nagarakatte

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2012

Milo M. K. Martin, Associate Professor of Computer and Information Science
Supervisor of Dissertation

Val Tannen, Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Rajeev Alur, Professor of Computer and Information Science

André DeHon, Professor of Electrical and System Engineering

Steve Zdancewic, Associate Professor of Computer and Information Science

Emery Berger, Associate Professor, University of Massachusetts Amherst

PRACTICAL LOW-OVERHEAD ENFORCEMENT OF MEMORY SAFETY FOR C PROGRAMS

This dissertation is dedicated to my parents.

Without them, this would not have been possible.

Acknowledgments

This dissertation is a direct result of constant support and encouragement from my parents who had more confidence in my abilities than I had, at times, in my ability. Apart from my parents, there are numerous people who have been instrumental in the growth of my research career and my development as an individual.

My adviser, Milo Martin has had a transformative influence on me as a researcher. I am fortunate to have worked with him for the last five years. Milo provided me the initial insights, the motivation to work on the problem, and eventually has nourished my ideas. He was generous with his time and wisdom. He provided me an excellent platform where I could excel. Apart from the research under him, he gave me freedom to collaborate with other researchers independently. I have also learned a great deal about teaching, presentations, and mentoring that will be amazingly useful in my future career.

Among the other faculty members at Penn, Steve Zdancewic was like a second adviser to me. Collaborating with Steve kept my programming language interests active and thriving. Amir Roth convinced me to join Penn. I had a great time working with Amir for the first two years. I would like to thank other faculty members—Rajeev Alur, Benjamin Pierce, Boon Thau Loo, Andreas Haeberlen, and Sudipto Guha—for their great courses, advice, encouragement and discussions on various topics.

I would like to thank my committee—Rajeev Alur, Steve Zdancewic, Andre DeHon, and Emery Berger—for carefully reading my dissertation, providing great feedback, useful advice and suggestions along the way. I would also like to thank Madanlal Musuvathi and Sebastian Burckhardt for mentoring during my internship at MSR and for the subsequent collaboration over the last few years. I would also like to thank my adviser at IISc, R. Govindarajan for encouraging me to pursue this path in my initial years.

Jianzhou Zhao and Andrew Hilton have been two of my closest collaborators. I learned a great deal about computer architecture, programming, and a passion for teaching from Drew. Jianzhou has been my primary collaborator from my first day at Penn. We did course projects, research projects, and papers together. He inspired and amazed me with his devotion to work, work ethics, and the constant focus on the task at hand. Other members of the safety project—Jianzhou Zhao, Christian Delozier, Peter Michael Osera and Richard Eisenberg—have provided me help, encouragement, and have spent time polishing and refining ideas together.

I also thank the past and the present members of ACG group—Tingting Sha, Andrew Hilton, Colin Blundell, Vivek Rane, Arun Raghavan, Christian Delozier, Sela Mador Haim, Abhishek Udupa, James Anderson, and Laurel Emurian—for the interesting reading groups, numerous discussions, and for attending my practice talks.

I have had many amazing friends here, who made graduate school pleasant. Numerous discussions with Arun Raghavan on a wide variety of topics—cricket, politics, books, research, and many others—was stimulating and fun. I also enjoyed spending time with other friends—Adam Aviv, Michael Greenberg, Marie Jacob, Changbin Liu, Annie Louis, Emily Pitler, Sudeepa Roy, and Wenchao Zhao. Thank you all.

I want to thank the LLVM community for building a robust and a modular compiler, answering our questions, and for the entire ecosystem. I also thank Mike Felker, who single handedly managed numerous things for me and many others in the department. I also want to thank all other faculty members at Penn, other students at Penn, Moore business office, and the CIS departmental staff for the various activities at Penn. I also want to thank Penn recreation for the great squash courts and my numerous squash partners: Rick Titlebaum, Richard Burgos, John Hansen, Vivek Rane, Riju Ray, Shivjit Patil, Christina Castelino, and many others. I had a great time playing squash with you all.

The dissertation research at Penn was also funded in part from grants by numerous funding agencies[1]: National Science Foundation (NSF) grants CNS-0524059, CCF-0644197, CNS-1116682, CCF-1065166, and CCF-0810947, Defense Advanced Research Projects Agency (DARPA)

[1]This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

contracts HR0011-10-9-0008, Office of Naval Research (ONR) award N000141110596, and donations from Intel Corporation.

Finally, I would like to thank my family for all their sacrifices, motivation, and encouragement. I would like to thank my parents for providing me good education throughout my childhood.

ABSTRACT

PRACTICAL LOW-OVERHEAD ENFORCEMENT OF MEMORY SAFETY FOR C PROGRAMS

Santosh Ganapati Nagarakatte

Milo M. K. Martin

The serious bugs and security vulnerabilities that result from C's lack of bounds checking and unsafe manual memory management are well known, yet C remains in widespread use. Unfortunately, C's arbitrary pointer arithmetic, conflation of pointers and arrays, and programmer-visible memory layout make retrofitting C with memory safety guarantees challenging. Existing approaches suffer from incompleteness, have high runtime overhead, or require non-trivial changes to the C source code. Thus far, these deficiencies have prevented widespread adoption of such techniques.

This dissertation proposes mechanisms to provide comprehensive memory safety that works with mostly unmodified C code with a low performance overhead. We use a pointer-based approach where we maintain metadata with pointers and check every pointer dereference. To enable compatibility with existing code, we maintain the metadata for the pointers in memory in a disjoint metadata space leaving the memory layout of the program intact. For detecting spatial violations, we maintain bounds metadata with every pointer. For detecting temporal violations, we maintain a unique identifier with each pointer. This pointer metadata is propagated with pointer operations and checked on pointer dereferences. Coupling disjoint metadata with a pointer-based approach enables comprehensive detection of all memory safety violations in unmodified C programs. This dissertation demonstrates the compatibility of this approach by hardening legacy C/C++ code with minimal source code changes. Further, this dissertation shows the effectiveness of the approach by detecting new memory safety errors and previously known memory safety errors in large code bases. To attain low performance overheads, this dissertation proposes efficient instantiations of this approach (1) within a compiler, (2) within hardware, and (3) with a hybrid hardware accelerated compiler instrumentation that reduces the overhead of enforcing memory safety, and thereby enabling their use in deployed systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The C programming language was originally designed to be a simple and efficient language with a lightweight runtime for the purpose of writing operating systems. As a result, the C programming language was assembly-like to harness the full performance of underlying hardware. Beyond the original purpose of building the UNIX operating system, C became widely popular because it provided the programmer the ability to get performance similar to hand-written assembly code while being higher level than assembly. Performance was crucial as the computing resources were at a premium during C's initial years.

As C gave primary importance to performance, a large number of software projects that were performance critical were written in C. An entire ecosystem of libraries and utilities emerged. This enabled languages like C and its variants (C++, Objective-C, CUDA, and many others) to become the gold standard for implementing a wide range of software systems from low-level system software (operating systems, virtual machine monitors, language runtimes, and embedded software) to performance-critical software of all kinds. Features such as low-level control over memory layout, explicit manual memory management (*e.g.*, *malloc()* and *free()*), and proximity to the underlying hardware made C the dominant language for many domains. Further, C and its variants also became popular in the parallel programming world with its focus on performance (Cilk [55], Unified Parallel C [26], C++, and CUDA). With its primary focus on performance and a rich ecosystem, C and its variants are the among the most widely used languages even today with majority of the client-side applications written in them. Altogether such systems comprise millions of lines of C code, preventing the complete transition away from C or its variants anytime soon.

A key drawback of C is that it does not ensure that the programmers use many of its low-level performance oriented features correctly and safely. The lack of memory safety is one such major drawback. Informally, memory safety is the property that ensures that all memory accesses adhere to the language specification (*i.e.*, all accesses to only allocated memory and within the prescribed object bounds). Memory safety violations in C arise as a result of two reasons: (1) accessing memory locations beyond the allocated region for the object or an array, which is termed as a *spatial memory safety* violation (also known as buffer overflows) and (2) accessing memory locations that have already been deallocated (typically while managing memory manually), which is termed as a *temporal memory safety* violation (also known as dangling pointer errors). Although the reason for the lack of memory safety is perhaps not fundamental, ensuring memory safety requires checking that can have performance overheads and is hard to perform in the presence of C's weak typing and other features (as described in Chapter 2). Further, avoiding temporal safety violations with automatic memory management using a garbage collector requires a sophisticated runtime. These are the likely reasons that made C originally eschew such checking.

Without memory safety, seemingly benign program bugs anywhere in the code base can cause silent memory corruption, difficult-to-diagnose crashes, and incorrect results. For example, storing arbitrary values at an out-of-bound array location can potential overwrite other data structures. Suppose if the overwritten location holds a function pointer, then the program can jump to arbitrary memory locations. The memory locations that have been overwritten using a memory safety error may not be used immediately but may be used sometime later in the program after executing millions or billions of instructions. Debugging such memory corruption arising out of memory safety errors can be a nightmare. Further, the manifestation of such memory corruption can be dependent on the way data objects are laid out in memory. This non-determinism makes the problem of diagnosing the root cause of memory corruption challenging.

Worse yet, the lack of memory safety is the root cause of multitude of security vulnerabilities, which result by exploiting a memory safety error with a suitably crafted input. The buffer overflow vulnerabilities, use-after-free vulnerabilities, and other low-level vulnerabilities resulting as a lack of memory safety have compromised the security of the computing ecosystem as a whole [3, 7, 38, 107, 108, 127, 136]. We will describe some of the recent notable security vulnerabilities that have exploited memory safety errors in Section 2.1.3 of Chapter 2. These vulnerabilities are particularly nefarious because (1) they can occur anywhere in the code (not just in those parts deemed security-critical) and (2) they break the abstractions of the programming language. Such vulnerabilities allow the attacker to manipulate the program into doing their bidding, either by injecting binary code directly into the program or by manipulating the program into making malicious system calls to install malware and otherwise completely compromise the system. Memory safety vulnerabilities are found in all sorts of programs that are written in low-level languages, including: operating systems, virtual machine monitors, web browsers, databases, user applications, shared libraries, and embedded software.

In contrast, safe languages such as Java and C#, enforce memory safety using a combination of language restrictions (restricted casts), dynamic checking (checked array accesses and checked type casts), and automatic memory management (using garbage collection). We will describe the rationale and the mechanisms in Section 2.2.2 of Chapter 2. Thereby these safe languages completely prevent this entire class of bugs and security vulnerabilities [38]. To avoid memory safety errors with C, one alternative would be to port existing legacy C code to these safe languages. However, porting legacy C applications to safe languages is non-trivial. Further, these safe languages may not be appropriate in some domains.

Given that C is continuing to be relevant, this dissertation seeks to enforce memory safety for C and thus eliminate the entire class of bugs and security vulnerabilities that result from memory safety errors. In that process, this dissertation tries to answer the following questions. First, can C be made completely memory safe? Second, what checking needs to be performed to enforce safety? Third, is it possible to provide memory safety for existing C source code? Fourth, how can memory safety be enforced with low performance overheads?

This dissertation seeks to retrofit C with memory safety that satisfies the following goals (we call it the 3 C's of memory safety): (1) *comprehensiveness* - provide the ability to detect all memory safety errors, (2) *compatibility* - provide the ability to enforce memory safety without source code modifications, and (3) *competitive performance* - provide the ability to run on deployed systems with low performance overheads. In the rest of this chapter, we describe how the existing proposals fail to attain these goals (Section 1.1), and describe our pointer-based checking with disjoint metadata that can attain comprehensiveness and compatibility goals (Section 1.2). We also describe the instantiations of this approach within the compiler (Section 1.3), within hardware (Section 1.4), and with a hybrid hardware compiler runtime (Section 1.5) to attain the competitive performance goal.

1.1 Challenges with Existing Proposals

The security problems as a consequence of the lack of memory safety in C programs has a long history. Recognizing the importance of the problem caused by the lack of memory safety, there has been significant prior work on proposals for detecting memory safety errors [13, 14, 16, 20, 27, 29, 32, 35, 43, 44, 47, 51, 54, 73, 92, 94, 97, 103, 111, 141]. At one end of the spectrum are partial countermeasures that prevent some memory safety violations. Although, modern operating systems and compilers employ many partial countermeasures (*e.g.*, guarding the return address on the stack, address space randomization, non-executable stack), vulnerabilities still persist [31]. Although these are reasonably effective [117] at times, these countermeasures are responses to vulnerabilities created by the adversary. Moreover, new vulnerabilities [2, 3, 9, 10, 31, 108, 136] that defeat these countermeasures keep emerging resulting in an enduring arms race.

The fundamental cause of memory safety violations is that languages like C provide low-level features such as arbitrary type casts, array-pointer conflation, and manual memory management but do not ensure that the programmers use these features safely. Unlike strongly typed languages where the bounds of the object is readily available with the object, such metadata may not be readily available in C as its weakly typed. Hence, maintaining and propagating the appropriate metadata so that pointer dereferences can be checked is a crucial step in enforcing memory safety.

There are two main design choices that the checking approaches make to enforce memory safety: (1) what metadata to maintain and (2) where to maintain the metadata. The type of metadata determines the ability to detect various errors. The placement of the metadata determines the ability to work with existing code. Common approaches maintain metadata in one of the following ways: (1) metadata with pointer, (2) metadata with the object, and (3) metadata in a separate metadata space. We describe many approaches using these design choices in detail in Chapter 2. Many of the prior approaches are primarily debugging aids providing varying degrees of memory safety. Hence, the design choices chosen by them are different from what would be required to prevent security vulnerabilities in deployed code.

To summarize, all prior solutions suffer from one or more of the following problems: failure to detect all spatial errors (*e.g.* stack) and/or temporal errors (*e.g.* due to reallocation of memory), have high runtime overheads, and/or require significant changes to the source code. The above mentioned drawbacks have resulted in these techniques being restricted to debugging contexts, rather than being used all the time in deployed code. With recurring security vulnerabilities, the comprehensiveness in enforcing memory safety and the associated performance overhead of such enforcement matters. This shift in concerns guides the design of our approach to enforce memory safety.

1.2 Pointer-Based Checking with Disjoint Metadata

This dissertation builds on top of the prior pointer-based checking approaches for enforcing memory safety [16, 73, 94, 131], which detect memory safety violations at runtime. In a pointer-based checking approach, metadata is maintained with pointers and all pointer dereferences are conceptually checked. Prior pointer-based checking approaches change the representation of the pointer into a multi-word fat pointer. The use of a fat pointer changes the memory layout in programmer visible ways thereby requiring (1) source code changes in presence of arbitrary type casts, and (2) metadata marshaling while interacting with external libraries. We describe the problems with fat pointers in detail in Chapter 2. In contrast to prior pointer-based approaches that maintained metadata with the pointer using a *fat* pointer, our approach maintains the pointer metadata in a disjoint metadata space. The use of disjoint metadata enables us to revisit pointer-based checking generally considered invasive for retrofitting memory safety for real world C applications.

Separating the pointer and the metadata in memory requires an efficient mapping between the pointer and its disjoint metadata. We use the address of the pointer (note it is different from the address that the pointer points to) to load/store the metadata when the pointer is being loaded/stored using table lookups. The use of disjoint metadata with a pointer-based approach ensures that (1) arbitrary type casts do not manufacture metadata providing comprehensive detection and (2) provides compatibility as memory layout of the program is not changed. However, disjoint metadata requires table lookups to obtain metadata for pointers, which can result in performance overheads.

To provide memory safety, we need to detect both spatial and temporal safety violations. To provide spatial safety, bounds metadata is associated with pointers when pointers are created. The bounds metadata is propagated on pointer operations and explicitly loaded/stored from the disjoint metadata space on pointer loads/stores. The pointer dereferences are checked to ensure that the pointer is within bounds. To provide temporal safety, we associate a unique identifier with the pointer that points to allocated memory on every memory allocation. The identifier is marked as invalid on memory deallocations. This identifier metadata is propagated with every pointer operation. A temporal safety check inserted conceptually before every pointer dereference, checks that the identifier metadata associated with the pointer is still valid. As identifiers are never reused, pointer-based approach with identifier metadata in a disjoint metadata space detects all temporal safety violations even in the presence of reallocations. Chapter 3 describes our approach in detail.

The use of disjoint metadata with pointer-based checking enables us to attain the comprehensiveness and compatibility goals. We accomplish the goal of competitive performance with an efficient implementation. In the next few subsections, we describe the instantiation of pointer-based checking approach with disjoint metadata within the compiler, within the hardware, and with hardware acceleration for the compiler instrumentation to attain low performance overheads.

1.3 Can Pointer-Based Checking be Performed within the Compiler?

As we described in the previous section, pointer-based checking approach with disjoint metadata can provide comprehensive and compatible memory safety. To provide low performance overheads, we perform the instrumentation within the compiler unlike prior approaches [16, 94, 131] that have used source-to-source translation. Our compiler instantiation of the pointer-based checking approach with disjoint metadata called *SoftBoundCETS* (unification of our prior work: SoftBound [92] and CETS [93]) operates on the intermediate representation (IR) of the LLVM compiler. To perform pointer-based checking within the compiler, *SoftBoundCETS* instrumentation needs to identify pointers. *SoftBoundCETS* instrumentation leverages the type information in the LLVM IR to identify pointers, and adds additional code only for operations that manipulate pointers. To shadow every word of memory for the disjoint metadata space, *SoftBoundCETS* uses a two-level trie data structure that is allocated on-demand. The two-level trie lookups are performed only when pointers are loaded/stored, which are identified using the type information in the IR. Thus, the use of type information minimizes the amount of added instrumentation code.

A compiler instrumentation also requires a mechanism to propagate metadata with function calls in an ISA independent manner. To propagate metadata for pointers arguments/return values with function calls, *SoftBoundCETS* uses a shadow stack that mirrors the call stack of the program. Unlike the call stack that holds the activation record, the shadow stack holds the metadata for the pointer arguments and return values for the frames in the call stack. Further, shadow stack provides a mechanism for dynamic typing between the arguments pushed at the call site and the arguments retrieved by the callee preventing memory safety errors with mismatched function signatures.

Finally, *SoftBoundCETS* further reduces the performance overhead by performing its instrumentation on optimized code, which is obtained after executing an entire suite of existing compiler optimizations. Our experimentation with SPEC benchmarks show that full memory safety can be enforced for unmodified C programs with 108% performance overhead on average. The *SoftBoundCETS* prototype is publicly available for download [6] and is also available in the experimental LLVM subversion repository. We have used the prototype with approximately one million lines of code detecting previously unknown (new) and known memory safety violations.

1.4 Can Pointer-Based Checking be Performed within the Hardware?

To implement pointer-based checking approach with disjoint metadata on mostly unmodified binaries with low performance overheads, this dissertation proposes a hardware proposal *Watchdog*. Unlike prior approaches for instrumenting binaries that have primarily used dynamic binary translation, *Watchdog* proposes instrumentation within the hardware using micro-operation injection. An optimized injection of micro-operations within the hardware avoids the huge performance overheads that is common with either dynamic binary instrumentation or binary translation.

To implement pointer-based checking on binaries, *Watchdog* needs to identify instructions manipulating pointers. The binaries typically do not provide any such information. In the absence of such information, *Watchdog* could perform additional instrumentation to perform checks, metadata accesses and propagation on each instruction. To mitigate such unnecessary operations especially the disjoint metadata access overheads, we propose two pointer identification schemes: (1) conservative pointer identification that treats every word sized load/store as a pointer load/store, and (2) extensions to the instruction set architecture to identify pointer loads/stores.

Once pointers are identified, *Watchdog* obtains information about memory allocations and deallocations from the runtime to associate bounds and identifier metadata with pointers. *Watchdog* injects micro-operations to (1) propagate the metadata with pointer operations, (2) check the metadata on memory accesses, and (3) access the metadata from the disjoint metadata space on pointer loads/stores. To streamline the implementation and reduce runtime overhead, *Watchdog* eliminates metadata copies among registers via modified register renaming and uses a dedicated metadata cache to reduce the checking overhead.

Further, *Watchdog* implements pointer-based checking with small localized changes to the hardware: (1) modified instruction decoder to inject micro-operations, (2) changes to the register renaming logic, and (3) extra hardware for the dedicated metadata cache. Using these hardware extensions, *Watchdog* reduces the overhead of providing full memory safety to just 24% on average for our benchmarks, which is an order of magnitude better than prior binary instrumentation and binary translation schemes.

1.5 Can Pointer-Based Checking be Performed with a Hardware/Compiler Hybrid?

From our experience building *SoftBoundCETS* instrumentation within the compiler and *Watchdog* hardware for instrumenting mostly unmodified binaries, we propose a synergistic hardware accelerated compiler instrumentation. We use the name *HCH* for this **h**ardware-assisted-**c**ompiler instrumentation **h**ybrid design. One of the bottlenecks with *SoftBoundCETS* instrumentation is the large number of instructions required to perform the checks and the metadata accesses. In contrast, identifying pointers precisely to minimize the amount of extra micro-operations is reasonably difficult with *Watchdog* as binaries lack such information.

We alleviate both these concerns in the hybrid design *HCH* by (1) leveraging the compiler to perform the instrumentation using the type information in the LLVM IR and (2) accelerating the checks and the metadata accesses using new ISA instructions provided by the hardware. We propose four new instructions to be added to the ISA: (1) a spatial check, (2) a temporal check, (3) a metadata load and (4) a metadata store. We explore variants of these instructions by leveraging the wide vector registers in modern processors. We find that the use of vector registers can reduce the register pressure that arises in the presence of pointer metadata. Hence, using vector registers with the new instructions is a promising option for accelerating pointer-based checking with disjoint metadata. Using the new ISA instructions along with the compiler instrumentation, we are able to reduce the performance overhead of the hybrid scheme to 39% on average with simpler hardware for our benchmarks.

1.6 Contributions of this Dissertation

This dissertation makes the following contributions:

- **Proposes the use of disjoint metadata with pointer-based checking to provide spatial and temporal safety.** Prior approaches that explored pointer-based checking primarily used inline metadata with pointers, which required complicated analyses to protect the metadata. The use of disjoint metadata as proposed in this dissertation leaves the memory layout of the program unchanged while ensuring the integrity of the metadata. Thus, disjoint metadata enables a simple local instrumentation to provide comprehensive and compatible memory safety. This approach has been recently adopted in the Intel's production compiler (see Section 8.2 of Chapter 8 for details) [57].

- **Proposes the use of type information in the compiler to perform pointer-based checking.** Unlike prior pointer-based checking schemes that have source-to-source translations, our *SoftBoundCETS* instrumentation is structured within the compiler and leverages the type information within the compiler to minimize the instrumentation. Use of type information to identify pointers and minimize instrumentation overheads enables us to revisit pointer-based checking with simple local instrumentation.

- **Proposes hardware micro-operation injection to perform pointer-based checking on mostly unmodified binaries.** The injection of micro-operations with *Watchdog* streamlines the implementation of pointer-based checking by (1) accelerating the checks and the metadata accesses, and (2) avoiding the inefficiencies (such as spills and restores) resulting as a consequence of small number of registers available to the compiler.

- **Proposes new instructions to accelerate a hardware-assisted compiler instrumentation.** We propose the use of the hardware-assisted compiler instrumentation where the type information available to the compiler is leveraged to minimize instrumentation and new instructions are used to accelerate the checks and the metadata accesses further minimizing the hardware required to accelerate pointer-based checking. We propose the use of vector register in modern processors along with new instructions as an attractive option for reducing the performance overheads of pointer-based checking.

- **Quantitatively evaluates pointer-based checking with disjoint metadata with above three design points.** We evaluate pointer-based checking with our *SoftBoundCETS* prototype with real world applications to determine its ability to (1) detect all memory safety errors, (2) handle real world code, and (3) instrument code with low performance overheads. We evaluate the hardware extensions and the hybrid using an x86 simulator. Our experiments indicate that pointer-based checking with disjoint metadata can enforce memory safety with low performance overheads without requiring source code changes enabling it to be deployed in live systems.

1.7 Dissertation Structure

The dissertation can be broadly organized into three parts: first, background on enforcing memory safety (Chapter 2); second, our proposal for enforcing memory safety using a pointer-based checking approach with disjoint metadata (Chapter 3); and third, our instantiations of the pointer-based checking with disjoint metadata within the compiler, in hardware, and with a hardware-compiler hybrid instrumentation (Chapters 4, 5, and 6).

Chapter 2 provides background on enforcing memory safety, closely related approaches for enforcing memory safety, and problems with existing approaches in the presence of arbitrary casts. Chapter 3 describe the basic checking approach where we maintain metadata with pointers in a disjoint metadata space, and it also describes how the use of disjoint metadata enables the pointer-based checking approach to detect all memory safety errors while requiring no source code changes. Chapter 4 provides a background on the LLVM compiler, describes our *SoftBoundCETS* instrumentation that leverages the type information in the LLVM IR, describes the shadow stack mechanism to propagate metadata for pointer arguments and returns with function calls, and evaluates the performance overheads of the *SoftBoundCETS* prototype. Chapter 5 describes *Watchdog*, our hardware instantiation of the pointer-based checking approach. The chapter describes the mechanisms used to identify pointers, and inject instrumentation within the hardware. Chapter 6 describes the bottlenecks with both pure compiler and hardware instrumentation. Further, the chapter describes the new instructions added to accelerate the compiler instrumentation. Chapter 7 describes the other related work that indirectly enforce memory safety. We conclude in Chapter 8 by summarizing our proposals and expressing our opinions on future challenges and barriers in the universal adoption of memory safety enforcement techniques.

1.8 Differences from Previously Published Versions of this Work

This dissertation builds upon the previous published results of Nagarakatte *et al.* [91, 92, 93]. The key idea of pointer-based checking with disjoint metadata for the compiler-based instrumentation is derived from our PLDI 2009 paper on providing spatial safety [92]. The presentation of the pointer-based checking approach in this dissertation extends prior work [92, 93] in three directions: (1) unifies both spatial and temporal safety enforcement mechanisms, (2) provides a consolidated and unified discussion on how disjoint metadata enables the pointer-based checking approach to provide comprehensive and compatible memory safety, and (3) provides a streamlined implementation, which has more optimizations and is more robust, in a new LLVM compiler. The compiler instrumentation for providing spatial and temporal safety were proposed earlier in our earlier papers [92, 93]. The *SoftBoundCETS* compiler instrumentation described in this dissertation provides a new mechanism for propagating metadata for pointer arguments with function calls using a shadow stack. Further *SoftBoundCETS* is evaluated comprehensively highlighting the benefits of compiler-based instrumentation with a more robust and streamlined prototype. The presentation of the *Watchdog* chapter in this dissertation integrates bounds checking along with use-after-free checking proposed in our prior work [91]. Chapter 6 describes our new results in enforcing memory safety using a hardware-assisted compiler instrumentation hybrid.

Chapter 2

Overview of Memory Safety Enforcement

This chapter provides an overview of memory safety and the prior techniques that have been proposed to enforce memory safety. The goal of this chapter is provide background, terminology, and context to understand the rest of the dissertation. The problem of providing memory safety for C is a well researched topic and a large number of proposals that provide partial or complete memory safety have been proposed [15, 16, 20, 32, 43, 45, 46, 47, 51, 60, 67, 73, 75, 92, 93, 94, 103, 116, 131]. In this chapter, we will describe approaches that are essential to understand the dissertation's contribution. We defer the discussion of other less related prior work that indirectly provide memory safety [1, 5, 11, 12, 13, 17, 27, 29, 31, 88, 99, 100, 123, 132, 135] to Chapter 7.

We describe the problem of memory safety violations in C and their two kinds: spatial safety violations and temporal safety violations in Section 2.1. We discuss the dimensions involved in enforcing memory safety in Section 2.2. We present the essence of three broad approaches to detect spatial violations in Section 2.3 and temporal violations in Section 2.4. In Section 2.5, we describe the instrumentation mechanisms for enforcing memory safety and their tradeoffs. Finally, we present the challenges and inefficiencies with existing approaches and close the chapter with a summary.

Spatial error

```
int *p, *q, *r;
void foo(int i) {
 p = malloc(8);
 ...
 q = p + i;
 ...
 *q = ...;
 ...
}
```

Heap based temporal error

```
int *p, *q, *r;
p = malloc(8);
...
q = p;
...
free(p);
r = malloc(8);
...
... = *q;
```

Stack based temporal error

```
int* q;
void foo() {
  int a;
  q = &a;
}
int main() {
  foo();
  ... = *q;
}
```

Figure 2.1: Examples of spatial and temporal safety violations both on the stack and the heap.

2.1 The Problem of Memory Safety in C

The C programming language was originally designed to write low-level systems code as it provides proximity to the underlying hardware, control over the memory layout, and minimal runtime support. However, several features of C, such as the use of pointers, pointer arithmetic, unsafe type casts, pointers to middle of objects, its conflation of arrays and singleton pointers, and unchecked array indexing can cause simple programming mistakes to corrupt the values of arbitrary memory locations eventually breaking fundamental programming abstractions. Such memory corruption can inadvertently cause a program to crash either immediately or in a non-deterministic manner, produce wrong results, and can be the root cause of multitude of security vulnerabilities. Moreover, such memory corruption can cause crashes at a later instant of time making debugging extremely difficult. Accessing and corrupting the values of arbitrary memory locations beyond the memory locations that are allowed by the programming abstractions is termed as a *memory safety violation*. Memory safety violations occur as a result of violation of type safety in the program, unchecked array accesses, and unsafe manual memory management. The relationship between providing memory safety and ensuring type safety is discussed in Section 2.2.

2.1.1 Spatial and Temporal Safety Violations

A memory safety violation could be either a *spatial safety* violation or a *temporal safety* violation. A spatial safety violation occurs when a program accesses a memory location outside the bounds of the object associated with the pointer or an array. Spatial safety violations include bounds violations, accessing uninitialized pointers, dereferencing NULL pointers and pointers manufactured using arbitrary casts from integers. Bounds violation arise as a result of pointers pointing to memory locations outside the bounds of the allocated object through arbitrary pointer arithmetic and type casts. Further, conflation of pointers and arrays in C makes it easier for programmers to erroneously access out-of-bound locations. Similarly uninitialized pointers can have arbitrary values (as they are uninitialized) and therefore can access arbitrary memory locations. A NULL pointer when combined with pointer arithmetic can overwrite and corrupt arbitrary memory locations.

A temporal safety violation occurs when a pointer is used to access a memory location after the object has been deallocated. Temporal safety violations include: dangling pointer dereferences (referencing an object that has been deallocated) and invalid frees (calling *free()* with a non-heap address or pointer to the middle of a heap allocated or freeing already freed pointer). Using a dangling pointer, a program can write to deallocated memory locations which could have been potentially reallocated to other objects thereby corrupting the values at arbitrary memory locations. On the other hand, invalid frees are not temporal memory safety violations per say but can corrupt the metadata maintained by the memory manager causing arbitrary memory corruption.

Figure 2.1 shows examples of common kinds of memory safety errors, including a spatial violation and a temporal safety violation. The left-most example in Figure 2.1 on page 13 demonstrates a spatial error where pointer `q` is dereferenced beyond the bounds. The examples on the right in Figure 2.1 on page 13 show dangling pointer errors involving the heap and stack. In the example in the middle, freeing `p` causes `q` to become a dangling pointer. The memory pointed to by `q` could be reallocated by any subsequent call to *malloc()*. In the right-most example in Figure 2.1, `foo()` assigns the address of stack-allocated variable `a` to global variable `q`. After `foo()` returns, its stack frame is popped, thereby `q` points to a stale region of the stack, which any intervening function call could alter. In both cases, dereferencing `q` can result in garbage values or data corruption.

2.1.2 Why are Memory Safety Violations Common with C?

There are multiple features in C code that can cause simple programming errors to result in memory safety violations. The fundamental reason behind memory safety violations is that C provides many features such as weak typing that allow arbitrary type casts, array-pointer conflation, and manual memory management but does not provide mechanisms (either static or dynamic) that check whether programmers use these features in a safe manner. Further, many of C's features makes the job of checking the usage difficult. Some examples include: (1) the lack of singleton type that requires every memory access to be checked, and (2) weak typing with unsafe type casts loses all the information about object sizes. As a result, a *void* * in the C code can be: (1) a pointer to an object allowed by the language specification (arrays, structures, singletons, sub-fields in a structure), (2) an out-of-bounds pointer, (3) a dangling pointer, (4) a *NULL pointer*, (5) an uninitialized pointer, and (6) a pointer manufactured from an integer. To check whether a pointer dereference is valid according to the language specification, appropriate metadata should be maintained with the pointer. Maintaining the appropriate metadata and checking the maintained metadata before a pointer dereference makes the problem of enforcing correct usage hard.

2.1.3 Consequences of Memory Safety Violations

The serious bugs and security vulnerabilities facilitated by C's lack of memory safety are well known. The lack of memory safety leads to bugs that cause difficult-to-diagnose crashes, silent memory corruption, and incorrect results. Worse yet, it is the underlying root cause of a multitude of security vulnerabilities [3, 4, 7, 8, 10, 31, 38, 49, 71, 72, 107, 107, 108, 127]. These violations allow an attacker to corrupt values in memory [31], inject malicious code, and initiate return-to-libc attacks [107] breaking fundamental programming abstractions. These memory safety violations are particularly insidious because even small coding mistakes anywhere within a large codebase can result in a catastrophic compromise of the entire system.

Both spatial and temporal safety violations have been used as the root-cause of security vulnerabilities. For one example, in November 2008 Adobe released a security update that fixed several serious buffer overflows [8]. Attackers have reportedly exploited these buffer-overflow vulnerabilities by using banner ads on web-sites to redirect users to a malicious PDF document crafted to take complete control of the victim's machine [7]. For another example, as of March 2009, millions of computers worldwide were infected with the Conficker worm, which spreads primarily via a buffer-overflow vulnerability [108].

Security vulnerabilities exploiting spatial memory safety violations have a long history. On the other hand, vulnerabilities exploiting temporal safety violations are becoming increasingly common in the recent years. Temporal safety violations have been used in real-world attacks [49, 71, 72] and many such vulnerabilities have been disclosed recently, including several in mainstream software such as Microsoft's Internet Explorer [3], Adobe's Acrobat Reader, Firefox [10], and OpenSSL [4]. In one notable incident, a use-after-free vulnerability was exploited in the widely publicized attack on Google and other companies in early 2010 [2, 3].

2.2 Enforcing Memory Safety

Memory safety violations and the concomitant security vulnerabilities can be avoided completely by moving to safe languages such as C# and Java or by enforcing memory safety for C. In the next few subsections, we describe the relationship between type safety and memory safety, and describe how safe languages provide memory safety using a combination of type safety and dynamic checking. Subsequently, we revisit how memory safety can be enforced for C with dynamic checking, and describe the trade-offs in various domains with dynamic checking.

2.2.1 Relationship Between Type Safety and Memory Safety

A type system in a statically typed language is a static analysis technique that can be used to prove the absence of certain program behaviors by classifying the expressions in the language according to the values they compute. A type system calculates a static approximation of the runtime behaviors of the expressions in the program. Like any static analysis technique, they are conservative and can reject programs that are well behaved during the execution. In contrast to static type systems, type tags can be maintained and checked at runtime. These runtime type tags dynamically check that the expressions have the correct type tags in accordance with the values they compute.

Type safety is closely linked to memory safety. To provide type safety, memory safety needs to be enforced apart from ensuring the proper use of abstract data types in the program. Memory safety needs to be enforced to provide type safety due to the following reasons: (1) spatial safety violations can break type safety (for example, unchecked array accesses can corrupt arbitrary memory locations breaking type safety) and (2) temporal safety violations similarly can break type safety (accesses to deallocated memory locations that have been potentially reallocated to a totally different data structure can corrupt arbitrary data structures breaking type safety). In contrast to the type safety enforcement that implicitly provides memory safety, memory safety enforcement does not necessarily imply type safety. A memory-safe program is not necessarily type safe as memory safety only ensures that each pointer dereference accesses a memory location that is legal to access according to the language specification but memory safety does not necessarily ensure that runtime values of expressions are consistent with their types.

As memory safety violations can easily break the type safety guarantees and statically checking that program is memory safe without false positives is infeasible, a program still needs extra dynamic checking to ensure memory safety. However, the type information in the program can be leveraged to reduce the amount of memory safety checking performed. In the next section, we will describe the goals in performing dynamic checking to enforce memory safety.

2.2.2 Memory Safety with Safe Languages

Safe languages such as Java and C# enforce memory safety using a combination of strong typing, extra runtime checks, and automatic memory management. Safe languages avoid temporal safety violations by relinquishing manual memory management and relying on garbage collection. To provide spatial safety, these language rely on the type system and the runtime bounds checks. However, performing such checks is easier than for C as the type system explicitly distinguishes pointers (references) to objects from arrays. Further, these languages allow only certain type casts (*i.e.* type cast between objects in the same class hierarchy using upcasts and downcasts), and it dynamically checks that these type casts are valid using runtime tags during downcasts. The upcasts move the object up in the class hierarchy. Hence, upcasts cannot cause a spatial safety violation. A strong type system coupled with runtime checks on array access and downcasts enable safe languages to prevent spatial safety violations.

2.2.3 Design Alternatives in Enforcing Memory Safety for C

There are three broad alternatives to retrofit memory safety to C. One alternative is to restrict the language to disallow all the idioms that make the task of enforcing memory safety harder as described in Section 2.1.2. To enforce memory safety in this manner, one would have to disallow pointer type casts, provide explicit identification of singletons and pointers, avoid manual memory management and check array accesses at runtime similar to safe languages. However, such restrictions would likely rule out a large number of C programs and may be applicable in only a few domains.

The second alternative would be to use a static analysis to check that the program is memory-safe. A static analysis can prove that each memory accesses is safe with respect to memory safety for all inputs. Hence such an analysis will not incur any performance overhead once the analysis is performed. However, the key problem with such static analyses is the numerous false positives (erroneous classification of programs to be memory unsafe when are indeed memory-safe). As precisely identifying memory unsafe accesses is equivalent to solving the halting problem, recent proposals [22, 48] have explored improving the precision by either restricting the classes of memory safety errors detected in the program (such as array indexing and string manipulation [48]) or by leveraging extra information in the program and using abstract interpretation [22]. Distinguishing false positives from the real memory safety errors is a tedious and time consuming task, many static analysis techniques have sacrificed soundness and have been targeted towards specific kinds of bugs.

The third alternative is to perform dynamic checking. A dynamic checking technique would need to maintain metadata and check the metadata on pointer accesses. Any dynamic checking technique that attempts to retrofit memory safety has to make tradeoffs and design choices along three dimensions of completeness, compatibility and performance overhead. First, completeness determines the ability of the technique to detect all memory safety errors without any false positives. As even simple errors could be exploited as security vulnerabilities, any scheme would ideally want to achieve completeness. Second, compatibility determines the ability of the technique to work with existing and legacy code. Significant restrictions to the language requires a significant rewrite of the existing code, thereby preventing adoption due to incompatibility with existing source code. Third, the runtime performance overhead determines the cost of enforcing memory safety with dynamic checking. As untested inputs can have memory safety violations, the memory safety enforcement techniques need to be adopted even in deployed systems. To encourage their adoption in deployed systems, all techniques strive to reduce the runtime performance overhead.

2.2.4 Bug Finding vs Always-on Dynamic Checking Tools

Dynamic checking techniques that enforce memory safety can be broadly classified into two categories based on their goals: (1) bug-finding tools and (2) always-on enforcement tools. Bug finding tools are primarily employed during the development stages (testing/debugging) with the goal of detecting memory safety violations. They are generally best-effort detection techniques without false positives. As they are not necessarily employed in the released software runs, performance overheads tend to be higher. On the other hand, always-on checking seek to be deployed even in the released software runs as the program may have memory safety violations with different inputs and these untested inputs can lead to security vulnerabilities. Having low performance overheads is important for always-on enforcement tools, which in-turn is important to detect security vulnerabilities.

Beyond completeness and performance overheads, compatibility with existing C code without requiring source code changes is also desirable. As we discuss later in Section 2.3.4, retrofitting memory safety while supporting all the idiosyncratic features of legacy C code to maintain compatibility could add additional overhead. The next few subsections describe and compare the approaches that detect memory safety violations, focusing on their performance, completeness, and compatibility attributes.

2.3 Detecting Spatial Safety Violations

The techniques that detect and prevent spatial safety violations can be classified into three categories described below: tripwire approaches, object-based approaches, and pointer-based approaches.

2.3.1 Tripwire Approaches

Tripwire approaches place a *guard block* of invalid memory between memory objects. The guard block prevents contiguous overflows caused by walking past an array boundary with a small stride. Although small strides will hit the guard block flagging spatial violations, a larger stride can jump over the tripwire and access data from another object undetected, causing a spatial safety violation. Hence these schemes are not complete.

The tripwire approaches are generally implemented by tracking a few bits of state for each byte in memory; the additional bits indicate whether the location is currently valid [65, 97, 110, 126, 133]. When the memory is allocated, these bytes are marked as valid. Every load or store is instrumented to check the validity of the location.

Purify [65] and Valgrind's MemCheck [97] implement the tripwire approach using binary rewriting, but their large performance overhead restricts their use. Address Sanitizer [115] uses compiler-based instrumentation reducing their overheads to approximately 2×. Yong *et al.* [133] use static analysis and check only memory writes to reduce the runtime overhead of this approach to under 2× in many cases. Further, hardware can used to reduce the overhead by either using invalid ECC signatures to encode invalid memory locations [110] or adding hardware support for updating and checking the valid/invalid blocks [126]. Although useful for finding spatial violations (and many temporal safety violations as discussed in Section 2.4), a significant drawback of these schemes is that they cannot guarantee the detection of all spatial violations.

2.3.2 Object-Based Approaches

Object-based approaches [40, 44, 51, 75, 113] are based on the principle that all pointers are properly derived pointers to their intended referent (the object they point to). Hence these approaches check pointer manipulations to ensure that the resultant pointer points to a valid object. To perform such checking, these approaches track all allocated regions of memory in a separate data structure. This data structure maps any location inside of an allocated region to the bounds information associated with the corresponding object. Pointer manipulation operations are checked to ensure that they remain within the bounds of the same object. The distinguishing characteristic of this approach is that bounds information is tracked per object and associated with the location of the object in memory, *not* with each pointer to the object. Every pointer to the object therefore shares the *same* bounds information.

The object-based approach has at least two important advantages. First, the memory layout of objects is unchanged, which improves source and binary compatibility. Second, all heap memory allocations (*i.e.*, calls to *malloc()*) update the object-lookup data structure, which allows every valid pointer to be mapped to an object, even if it was allocated by un-instrumented code. This behavior allows object-based schemes to transparently interact with legacy libraries that have not been instrumented, therefore improving the overall compatibility of the system.

However, the object-based approach has three disadvantages. First, out-of-bounds pointers, which are allowed by the C language specification as long as they are not dereferenced, require special care. Moreover, such pointers are relatively common in existing C code [75]. Object-based schemes use special out-of-bounds proxy objects [44, 75] when out-of-bound pointers occur. If an out-of-bound pointer is modified so that it is back in bounds, this proxy object is used to

Sub object overflow

```
struct {
  char id[8];
  int account_balance;
} bank_account;

...

char * ptr = &(bank_account.id);
strcpy(ptr, "overflow......");

...
```

Figure 2.2: Example of a sub-object overflow.

recreate a valid pointer to the original object. Second, the object-lookup is a range lookup: a pointer to anywhere in the object must correctly map to the object's bounds information. This range lookup is often implemented as a splay tree, which can be a performance bottleneck, yielding runtime overheads of 5× or more [51, 75, 113]. Subsequent proposals have considerably mitigated this issue, reducing the overhead of the object-table by checking only strings [113], using whole-program analysis to perform automatic pool allocation to partition the splay trees and eliminate lookup and checking for many scalar objects [40, 44], and by using efficient checks facilitated by enforcing allocation bounds (using binary buddy allocator) rather than precise object bounds [14].

The third significant drawback of the object-based approach is that its implementations are generally incomplete—they do not detect all spatial violations. For example, arrays inside structures are not always checked. To see why, consider a contrived example in Figure 2.2, where pointers to `bank_account` and `bank_account.id` are indistinguishable as they point to the same location and thus are associated with the same object-based bounds information. Hence the pointer `&(bank_account.id)` inherits the bounds of the whole object. When `ptr` is passed to `strcpy()` an overflow of `bank_account.id` can overwrite the rest of the `struct`, including the `struct`'s `account_balance` field—even if `strcpy()` is instrumented. Although contrived, this example demonstrates that sub-object overflows have the potential to result in serious bugs or security vulnerabilities.

Although object-based implementations have typically not targeted or addressed the detection of such sub-object overflows, some object-based proposals are more successful at preventing internal overflows. For example, SAFECode [44, 46], a recent instance of the object-based approach, uses whole-program static type analysis and type-homogeneous pool allocation to significantly improve coverage in such cases; it is capable of detecting many, but not all, such sub-object violations.

2.3.3 Pointer-Based Approaches

An alternative approach is the *pointer-based approach*, which tracks base and bound information with each pointer. This is typically implemented using a *fat pointer* representation that replaces some or all pointers with a multi-word pointer/base/bound. Such a fat pointer records the actual pointer value along with the addresses of the upper and lower bounds of the object pointed by the pointer. Two distinct pointers can point to the same object and have different base and bound associated with them, so this approach avoids the sub-object problem with object-based approaches discussed above. When a pointer is involved in arithmetic, the actual pointer portion of the fat pointer is incremented/decremented. On a dereference, the actual pointer is checked to see whether it is within the base and bound associated with it. Proposals such as SafeC [16], CCured [94], Cyclone [73], MSCC [131], and others [43, 95, 103] use this pointer-based approach to provide spatial safety guarantees. The pointer-based approach is attractive in that it can be used to detect sub-object overflows and avoid the extra mechanisms required to handle out-of-bound pointers. However, propagating and checking bounds for all pointers can result in significant runtime overheads.

To reduce these overheads, CCured [94] used whole-program type inference to identify pointers that do not require bounds checking. CCured classifies pointers into various kinds: *SAFE*, *SEQ*, and *WILD*. *SAFE* pointers have negligible performance overhead and are not involved in pointer arithmetic, array indexing, or unsafe typecasts. *SEQ* pointers are fat pointers that allow only pointer arithmetic and array indexing and are not involved in unsafe typecasts. *WILD* pointers allow arbitrary type casts, but require additional metadata and also any non-pointer store through a *WILD* pointer is required to update the additional metadata. This approach reduces the runtime overhead significantly, but CCured requires modifications to the source code to: (1) avoid introducing inefficient WILD pointers and (2) handle the memory layout incompatibility introduced by CCured's use of fat pointers.

```
struct A {
 size_t t1;
 void* ptr;
 size_t t2;
};

struct B {
 size_t f1;
 size_t f2;
 size_t f3;
 size_t f4;
};

void foo(struct A * p) {
 struct B * q;
 q= (struct B *)p;
 ...
 q->f3 = ...;
 ...
}
```

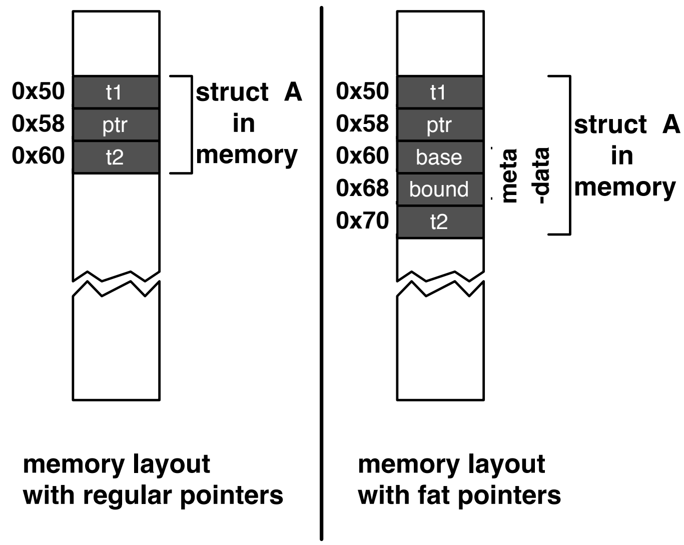


Figure 2.3: Memory layout changes in a program with fat pointers. The code snippet also shows how pointer involved in arbitrary type casts can overwrite the pointer metadata.

2.3.4 Source Incompatibility with Fat Pointers and Arbitrary Type Casts

The most significant disadvantage of the fat pointer-based approach is that fat pointers change memory layout in programmer-visible ways. This introduces significant source code compatibility issues requiring the source code to be modified [94]. Figure 2.3 illustrates the memory layout changes that occur with the use of fat pointers resulting in source incompatibilities. The code snippet in Figure 2.3 has a structure type *A* with three sub-fields: (1) an unsigned integer field (t1), (2) a void pointer field (ptr), and (3) another unsigned integer field (t2). In a normal program without fat pointers, this structure would be laid out in memory as shown in Figure 2.3, with each field adjacent to each other. With the use of fat pointers, the structure laid out in memory will have additional metadata along with original member fields as shown in Figure 2.3.

The first major problem with fat pointers is that it changes the memory layout in programmer visible ways. The modified memory layout makes interfacing with library code challenging. The libraries expect the structure fields to be at certain offsets, which will be different when the structure contains a fat pointer. To avoid such problems with external libraries, wrappers marshal the data structures with pointers. Such marshaling of data structures may require deep copies. Further, such deep copies of data structures are required even when the library just reads the data structures without performing any updates making the task of writing wrappers cumbersome while incurring performance overhead. To address this issue, attempts have been made to split the metadata from the pointer [94, 131]. These approaches partially mitigate some of the interfacing issues, but such techniques can increase overhead by introducing linked shadow structures that mirror entire existing data structures.

The second major challenge with the fat pointers is the protection of in-line metadata in the program. Arbitrary typecasts in the program can corrupt the metadata. The code snippet in Figure 2.3 presents a contrived yet feasible example of how the metadata maintained with fat pointers in memory can be corrupted in the presence of an arbitrary type cast. The function *foo* has a pointer argument that points to a structure of type *struct A*, which has a pointer field. The function creates a new pointer *q* by type casting the pointer *p* to be a pointer of type *struct B*, which has unsigned integer fields. Subsequent writes to the fields of *struct B* through *q* can overwrite the fat pointer metadata resulting in non-comprehensive detection of memory safety violations as shown in Figure 2.3.

To address metadata integrity problem, prior approaches have either forbidden the use of arbitrary type casts in the program or introduced extra metadata structures to protect the metadata in the presence of casts [94]. Forbidding unsafe type casts requires a rewrite of the existing code base losing source compatibility. Further many schemes ignore the problem with casts losing comprehensive protection. On the other hand, CCured uses WILD pointers that are dynamically typed to provide protection in the presence of casts. A WILD pointer in CCured comes equipped with a base field and a pointer field. The base field points to the start of a dynamically typed area, where there is a length field indicating the size of the area. At the end of the area, there is a set of tag bits that indicate which words in the dynamically typed area contain CCured's base pointers. To protect the metadata, every memory access through a WILD pointer must access the tag bits apart from the regular metadata. Using extra metadata structures to protect inline-metadata introduces significant performance overheads, as with WILD pointers in CCured, restricting the use of such schemes.

2.3.5 Comparison of Spatial Safety Approaches

Tripwires do not provide comprehensive detection. Further their overheads depend on the instrumentation method and implementation technique. Object-based and pointer-based approaches have complementary strengths and weaknesses in detecting spatial memory safety violations. Object-

<table>
<tr><th>Checking Taxonomy</th><th>Approach</th><th>Instrument. method</th><th>Runtime overhead</th><th>Metadata</th><th>Compatible with casts</th><th>Compre. Protection</th></tr>
<tr><td>Tripwire</td><td>MC [97]</td><td>Binary</td><td>$> 10\times$</td><td>Disjoint</td><td rowspan="5">Yes</td><td>No</td></tr>
<tr><td rowspan="4">Object based</td><td>J&K [75]</td><td rowspan="4">Compiler</td><td>$> 10\times$</td><td rowspan="4">Disjoint</td><td rowspan="4">No</td></tr>
<tr><td>Mudflap [51]</td><td>$> 40\times$</td></tr>
<tr><td>Baggy [14]</td><td>$1.1 - 2\times$</td></tr>
<tr><td>SAFECode [44]</td><td>$4\times$</td></tr>
<tr><td rowspan="6">Pointer based</td><td>SafeC [16]</td><td rowspan="4">Source</td><td>$> 10\times$</td><td rowspan="3">Inline</td><td rowspan="4">No</td><td rowspan="3">No</td></tr>
<tr><td>CCured – Safe/Seq [94]</td><td>$1.3\times$</td></tr>
<tr><td>CCured – Wild [94]</td><td rowspan="3">$2\times$</td></tr>
<tr><td>MSCC [131]</td><td>Disjoint*</td><td>Yes/No</td></tr>
<tr><td>MemSafe [120]</td><td>Compiler</td><td rowspan="2">Disjoint</td><td rowspan="2">Yes</td><td rowspan="2">Yes</td></tr>
<tr><td>Dissertation</td><td>Comp/HW</td><td>$1.1 - 2\times$</td></tr>
</table>

Table 2.1: Comparison of **spatial** safety approaches. The * indicates that the MSCC provides multiple implementations of the metadata. MSCC's high performance configuration does not use disjoint metadata losing comprehensive protection. This dissertation explores both compiler and hardware instrumentation approaches, and is represented by Comp/HW.

---

based approaches are highly compatible because they use a separate lookup tree for tracking object metadata, and thus they do not change the memory layout. In fact, they have been successfully applied to the entire Linux kernel [40]. However, object-based approaches cannot always enforce complete spatial safety because of sub-object overflows. In contrast, pointer-based approaches typically change the pointer representation and memory layout causing source code compatibility problems. Handling arbitrary casts is another important problem. For example, in CCured, arbitrary casts result in WILD pointers (which further complicate library compatibility) and may have significant performance ramifications. When whole-program analysis is applied to reduce the overhead of either scheme [44, 94], it can complicate the use of precompiled and dynamically loaded libraries.

Table 2.1 summarizes the various object-based and pointer-based approaches in contrast with the dissertation's approach. Object-based approaches such as Jones & Kelley [75] satisfy most of the attributes except for the detection of all sub-object violations. CCured with only *Safe/Seq* pointers has low overhead and is complete but lacks source code compatibility. MSCC [131] uses split metadata and run-time type information, but it has difficulties handling arbitrary casts and it does not detect sub-object overflows in the configuration with the lowest runtime overhead.

This dissertation proposes the use of disjoint metadata with pointer-based checking addressing the memory layout compatibility and arbitrary type cast problem while providing comprehensive detection of all errors. An arbitrary type cast can only manufacture pointers. The disjoint metadata is unchanged with such cast operations. This property enables us to handle casts and still provide comprehensive protection. Chapter 3 provides a detailed explanation of our approach. A recent proposal, MemSafe [120], builds on our idea of disjoint metadata to provide comprehensive protection.

2.4 Detecting Temporal Safety Violations

Beyond providing spatial safety, preventing temporal safety errors is important with an increase in the number of security vulnerabilities exploiting temporal safety violations. Temporal errors result as a consequence of unchecked manual memory management. With manual memory management in C, programmers allocate memory on the heap using the *malloc* function (also using similar allocation functions like *calloc*). Similarly, the programmers deallocate the memory using the *free* function provided by the runtime library. When the program erroneously dereferences a pointer that points to a deallocated memory (also called a dangling pointer), it can access/corrupt arbitrary memory locations resulting in temporal safety violations. Several approaches have been proposed to prevent and detect temporal safety violations with varying degrees of protection [16, 20, 45, 47, 65, 97, 101, 103, 126, 131]. We discuss few of the previously proposed closely related approaches in this section. Other related approaches are discussed in Chapter 7.

2.4.1 Garbage Collection in Safe Languages

Languages like Java, C#, ML and many others avoid temporal safety errors by relinquishing manual memory management. These languages use garbage collection. As garbage collection works on heap allocated storage, these languages disallow the allocation of objects on the stack (through language design). The programmer allocates the objects on the heap using the memory allocation routines. Garbage collector frees the unused objects when the program runs out of memory or reaches a memory threshold. When an object is no longer used and there are no references to the object, the garbage collector reclaims the allocated space. To perform such tracking, the garbage collector needs to locate all program locations that hold live pointer variables at runtime, and automatically free the unused locations. Further, these languages are type-safe, and can use accurate garbage collection. Although automatic memory management is widely using with safe languages, garbage collection is not always an answer in some domains. For example, non-deterministic pause times makes garbage collection unsuitable for real time systems.

2.4.2 Garbage Collection in Type-Unsafe Languages like C

An alternative to avoid temporal safety violations with C is to use garbage collection. However, as C is not type-safe, C can use only conservative garbage collection [24]. As with manual memory management, the programmer allocates memory using the garbage collection provided *malloc* function. However, deallocation operations using *free* do nothing. When the garbage collector kicks-in, it tracks all references contained in registers, on the execution stack, or nested inside other objects. The garbage collector reclaims memory that is unreachable. A conservative garbage collector [24] cannot move live objects in the heap as it does not know which memory location holds a pointer. This prevents the use of generational and compacting garbage collection algorithms that are widely used with type-safe languages. Further, conservative garbage collector can suffer from memory leaks due to integer values that accidentally appear like pointer addresses.

Allocating objects on the stack is common in C. To enable conservative garbage collection to work with C and prevent dangling pointers on the stack as illustrated in Figure 2.1 on page 13, all stack objects that can escape from the functions need to be allocated on the heap (heapification [94]). A conservative garbage collector when coupled with heapification can eliminate all dangling pointers in these programs. However, such a conservative garbage collector similar to garbage collectors with type-safe languages may not be appropriate for long running programs as it can leak memory. Further, non-deterministic pause times with garbage collector makes it unsuitable for writing low systems code, C's niche domain. Unlike our fault model where the program halts on a manual memory management error, garbage collector can tolerate memory management errors by masking them while preventing preventing security exploits. Hence, garbage collector may not be ideal in a debugging context as it masks errors rather than reporting them in a fail-stop fashion.

Alternatives to Garbage Collection for C As garbage collection is not ideal for low-level systems code written in C, manual memory management is still widely used. A temporal safety violation arises when the program dereferences a pointer that points to a deallocated memory location. Such pointers are also called dangling pointers. Memory writes through dangling pointers can corrupt arbitrary memory locations, which can become the root cause of security vulnerabilities. However, such temporal safety violations can be prevented by checking dangling pointer dereferences. We broadly call schemes that try to detect dangling pointer dereferences as temporal-checking schemes. We discuss the temporal checking schemes related to the dissertation in the following subsections.

2.4.3 Location-Based Temporal Checking

Location-based approaches (*e.g.*, [65, 75, 97, 126]) use the location (address) of the object to determine whether it is allocated or not. An auxiliary data structure records the allocated/deallocated status of each location. This data structure is updated on memory allocations (*e.g.*, *malloc()*) and deallocations (*e.g.*, *free()*). On a memory access, these auxiliary structures are consulted to determine whether the dereferenced address is currently valid (allocated) memory. As long as deallocated memory locations are never reallocated, this approach will detect all dangling pointer dereferences. However, if a location has been reallocated, this approach erroneously allows the pointer dereference to proceed—the information tracked by this approach is insufficient to determine that the pointer's original allocation area was freed and is now being used again for a potentially unrelated purpose. Thus, although such techniques are able to detect many dangling pointer dereferences, they cannot detect all temporal errors. These location-based approaches can also detect some spatial safety violations that access unallocated allocations beyond the allocated region.

There are two distinct ways of organizing the auxiliary data structures for recording the allocation information, each with different space/time trade-offs: (1) recording allocation ranges in a tree structure and (2) using a shadowspace to track the allocation status of each word in memory.

Tree-Based Location Lookup One approach to implementing the auxiliary data structure is to record all allocated regions of memory in a tree structure [75]. On a pointer dereference, a range lookup in the tree identifies whether the address pointed to by the pointer is currently allocated (when a mapping is found) or unallocated (when no mapping is found). The memory overhead of this approach is low because it is proportional to the number of live objects, but it requires a potentially slow range lookup (*i.e.*, splay tree walk) for each dereference check. Unlike the object-based approach to detect spatial safety violations that check pointer arithmetic, to detect temporal safety violations such schemes need to check dereferences.

Shadowspace-Based Location lookup An alternative approach is to use a *shadowspace* in which allocation/deallocation status is maintained with each byte (or word) of memory [65, 96, 126]. A shadowspace may be implemented as a large, directly accessed memory region, a hashtable [92], or a trie-based data structure [96, 138]. Accessing a shadow space entry is typically an efficient $O(1)$ operation, and thus provides fast checks. The memory usage is proportional to the size of the memory rather than just the number of allocated objects (as in the tree-based scheme), but tracking a bit per word is only a few percent memory overhead.

2.4.4 Identifier-Based Temporal Checking

An alternative approach is the *allocation identifier approach*, which associates a unique identifier with each memory allocation. Each allocation is given a unique identifier and identifiers are never reused. To ensure that this unique identifier persists even after the object's memory has been deallocated, the identifier is associated with pointers. On a pointer dereference, the system checks that the unique allocation identifier associated with the pointer is still valid. One implementation of pointer-based metadata is to expand all pointers into multi-word *fat pointers*. As described earlier in Section 2.3.3, fat pointers suffer from (1) compatibility problems as a result of memory layout changes, (2) library interfacing problems, and (3) the combination of fat pointers and arbitrary casts can lead to metadata corruption weakening the complete protection as illustrated in Figure 2.3 on page 23.

Set-Based Identifier Checking A set data structure (such as a hash table) is one approach for determining whether an allocation identifier is still valid (*i.e.*, the object has not been deallocated) [16]. The allocation identifier is inserted into the set during allocation (*e.g.*, *malloc()*) and removed from the set at deallocation (*e.g.*, *free()*), and thus the set contains an identifier if and only if the identifier is valid. Although set lookups can take just $O(1)$ time, performing a hash table lookup on every memory reference has the potential for introducing significant runtime overheads.

<table>
<tr><th>Checking Taxonomy</th><th>Approach</th><th>Instrument. method</th><th>Runtime overhead</th><th>Metadata</th><th>Compatible with casts</th><th>Complete protection</th></tr>
<tr><td rowspan="5">Location</td><td>MC [97]</td><td>Binary</td><td rowspan="2">10×</td><td rowspan="5">Disjoint</td><td rowspan="5">Yes</td><td rowspan="5">No</td></tr>
<tr><td>JK [75]</td><td>Compiler</td></tr>
<tr><td>LBA [29]</td><td rowspan="3">H/W</td><td rowspan="3">1.2×</td></tr>
<tr><td>SProc [60]</td></tr>
<tr><td>MTrac [126]</td></tr>
<tr><td rowspan="5">Identifier</td><td>SafeC [16]</td><td rowspan="3">Source</td><td>10×</td><td>Inline</td><td rowspan="4">No</td><td rowspan="5">Yes</td></tr>
<tr><td>P&F [103]</td><td>5×</td><td rowspan="2">Split</td></tr>
<tr><td>MSCC [131]</td><td>2×</td></tr>
<tr><td>Chuang [32]</td><td>Hybrid</td><td>1.2×</td><td>Inline</td></tr>
<tr><td>Dissertation</td><td>Compiler/HW</td><td>1.1x-2×</td><td>Disjoint</td><td>Yes</td></tr>
</table>

Table 2.2: Comparison of **temporal** checking approaches.

Lock-and-Key Identifier Checking To avoid a set lookup on each check, an alternative is to pair each pointer with two pieces of metadata: an allocation identifier—the *key*—and a *lock* that points to a location in memory called *lock location* [103, 131]. The key and value at the lock location will match if and only if the pointer is valid. A dereference check is then a direct lookup operation—a simple load from the lock location and a comparison with the key—rather than a hash table lookup. Freeing an allocation region changes the value at the lock location, thereby invalidating any other (now-dangling) pointers to the region. Because the keys are unique, a lock location itself can be reused after the space it guards is deallocated. As we use the lock and key method for providing complete memory safety, it is described later in detail in Chapter 3.

2.4.5 Analysis of the Temporal Checking Design Space

The general approaches described above have complementary strengths and weaknesses. Using identifiers permits detection of dangling pointer dereferences even if the memory has been reallocated to a new object. The disadvantages of identifiers stem from tracking per-pointer metadata, which adds potentially significant overhead to loads and stores of pointer values. Fat-pointer implementations also change data layout, which can reduce source compatibility and make interaction with libraries difficult.

Location-based checking is attractive because its disjoint metadata does not change the program's memory layout. By re-linking with a different *malloc()* library, this approach works even for objects allocated within libraries, and thus is highly compatible—using it requires fewer source program changes. However, because this approach tracks only allocations and does not maintain metadata with pointers, it does not detect pointers that erroneously point to reallocated regions of memory.

This dissertation adopts the identifier-based checking to detect all errors even in the presence of reallocations. To avoid the memory layout problems, it maintains the identifier metadata with pointers in a disjoint metadata space providing the best of both worlds: identifier-based checking and location-based checking.

2.5 Program Instrumentation

Given the approaches explained above, there are several different options for adding the checks necessary to enforce memory safety.

Binary Instrumentation One option is to instrument the program at the binary level, after compilation. Mechanisms for doing so range from completely static binary rewriting [65] (*i.e.*, produce a new executable from the old one) to partial emulation at the instruction level, intercepting control-flow transfers and interpolating new code [97]. The benefit of binary translation is that it operates on unmodified binary code and dynamically linked libraries, even when the source code is unavailable. However, the emulation and instrumentation overhead can contribute to high runtimes (10$\times$ slowdown is common [97]), in part because it is difficult to perform high-level optimizations. Further, typically binaries do not have much information and may be lacking the information (such as stack allocation sizes, pointers) to perform pointer-based checking. Moreover, these tools are also inherently tied to a specific instruction-set architecture.

Hardware-Assisted Instrumentation There are numerous ways in which the hardware can accelerate/implement the memory safety checking scheme. Hardware instrumentation schemes that augment the instruction execution with extra instructions to perform safety checking such as Memtracker [126], HardBound [43], LBA [29] are attractive as they capture desirable properties of binary instrumentation but with potentially lower runtime overhead and good backwards-compatibility. However, the key disadvantage is that such instrumentation requires new hardware. Further, such instrumentation is limited by the amount of information available in the binary which may be missing information such as allocation sizes and pointer information.

Source-Level Instrumentation The third option is source-level transformation to insert runtime checks (*e.g.*, [16, 94, 131]). This approach allows for use of source-code type information, and the resulting instrumented source is independent of any specific C compiler or instruction-set architecture. The instrumentation is applied before the code is optimized by the compiler (*e.g.*, before variables are register allocated or code is hoisted out of loops). Unfortunately, once the instrumentation code has been added, the additional memory operations introduced may limit the effectiveness of subsequent compiler optimization passes.

Compiler-Based Instrumentation Another option is for the compiler to instrument the code during compilation. This approach is similar to source-level instrumentation, but adds one key advantage that it can introduce instrumentation after the compiler has applied standard code optimizations. This reduces the amount of instrumentation code introduced and thereby reducing the performance overhead. Naively introducing safety checks before optimizations results in high performance overhead as the optimizer then would have to remove checks and instrumentation code, which is not necessarily straightforward.

2.6 Summary

In this chapter we provided a basic overview of memory safety enforcement techniques that provides the context for the rest of the dissertation. We described the goals and problems in retrofitting memory safety for C in Section 2.1. Any memory safety enforcement technique would ideally provide (1) completeness by detecting all memory safety violations, (2) compatibility with existing code, and (3) low or no performance and memory overhead. From the discussion of prior approaches, it was evident that pointer-based checking with appropriate metadata (bounds and identifier) can detect spatial and temporal safety violations. However, changing the representation of the pointer which changes the memory layout causes source code incompatibilities as discussed in Section 2.3.4. We observed that the high compatibility of object-based and location-based schemes is primarily attributed to the use of disjoint metadata. This motivates us to use disjoint metadata with pointer-based checking scheme to attain the above mentioned goals.

In the next chapter we describe our approach for providing complete memory safety using a pointer-based approach that maintains the bounds and identifier metadata in a disjoint metadata space providing completeness and compatibility. The subsequent chapters discuss the instrumentation strategies to reduce the performance overhead.

Chapter 3

Pointer-Based Checking with Disjoint Metadata

In this chapter we describe a pointer-based checking approach that maintains the metadata in a disjoint metadata space for enforcing memory safety. Subsequent chapters describe concrete instantiations of this high level approach within the compiler (Chapter 4), purely in hardware (Chapter 5), and with compiler assisted hardware support (Chapter 6). Although pointer-based checking [16, 73, 94, 131] has been proposed and investigated earlier, the primary advantage of our approach is that it maintains the metadata in a disjoint space that provides an opportunity to revisit pointer-based checking (generally considered invasive) for retrofitting C with practical memory safety. A practical scheme would enforce memory safety for existing programs without any source modifications. Hence, the primary goals of our approach are to retrofit memory safety for C by (1) detecting all the memory safety violations both spatial safety and temporal safety errors, (2) maintaining source compatibility with existing programs, and (3) performing such checking keeping the overheads low enough to be widely deployed in live systems.

3.1 Approach Overview

To meet the goal of comprehensive memory safety, we use a pointer-based approach where we maintain metadata with each pointer similar to prior schemes described in Chapter 2. However, one of the key advantages of our work is that it maintains the metadata in a disjoint metadata space, which leaves the memory layout of the program intact. The key challenge is to efficiently map a pointer to its disjoint metadata. After we narrow down our approach to a scheme that does pointer-based checking with disjoint metadata, the rest of the work is in streamlining the instrumentation to attain the required safety goals and performance targets. In a nutshell, each pointer has metadata that is assigned when pointers are created and propagated with pointer operations. To detect spatial safety errors, each pointer has bounds metadata. To detect temporal safety errors, each pointer has identifier metadata (organized as lock and key identifier).

The pointer's metadata is checked before memory accesses to detect memory safety errors. To detect spatial safety errors, a spatial check performed before a pointer dereference checks whether the pointer is within the bounds. To detect temporal safety errors, the identifier associated with the pointer being dereferenced is checked for validity. As unique identifiers, which are not reused, are allocated on memory allocations and the identifiers are marked as invalid on memory deallocations, temporal errors are detected even in the presence of reallocations as described in Section 2.4.4 of Chapter 2.

Pointer-based checking with disjoint metadata provides both binary compatibility (*i.e.* the library interfaces and data layout are unchanged) and provides good source compatibility (*i.e.* negligible to no source code changes), thereby enabling our approach to retrofit comprehensive memory safety to unmodified legacy C code base. We describe the conceptual operation of the pointer-based checking in the rest of this chapter. We conclude this chapter by demonstrating the compatibility and comprehensiveness of this approach by detecting previously known/unknown memory safety errors and by working with existing legacy code with negligible source code changes. We defer the efficient implementation of this approach in various parts of the tool chain to subsequent chapters: Chapter 4 describes an instrumentation on the intermediate representation of the compiler, Chapter 5 describes an instrumentation purely within the hardware, and Chapter 6 describes an instrumentation using hardware accelerated checks with compiler assistance .

3.2 Metadata with Pointers

Our approach maintains metadata with pointers. Hence we need mechanisms to identify pointers and maintain metadata with them. Pointers can be identified in one of the many ways: (1) directly from the information available in the source code, (2) using the type information in the typed intermediate representation within the compiler (assuming that such information is preserved by the compiler from the source code) as in Chapter 4, or (3) inferring all machine word (64-bit) sized integer data as pointers as in Chapter 5. To simplify exposition, we assume perfect identification of pointers in this chapter.

New pointers in C are created in two ways: (1) explicit memory allocation (*i.e. malloc()*) and (2) taking the address of a variable using the `&` operator. Whenever such pointers are created, our approach creates and associates metadata with each pointer by inserting additional operations. Our approach maintains separate pieces of metadata with each pointer to detect spatial and temporal safety violations as described in the following sections unlike prior approaches that maintain same metadata for both [120]. Further, maintaining different pieces of metadata enables us to optimize the spatial checks and temporal checks separately in our implementations of these schemes as described in Section 4.3 of Chapter 4.

3.3 Spatial Memory Safety Metadata and Checking

For detecting spatial violations, we maintain two pieces of metadata with each pointer, (1) the base and (2) the bound of the memory area pointed by each pointer. The base and bound metadata is created whenever a pointer is created on memory allocations using *malloc()* and by taking the address of a stack or a global variable using the address-of `&` operator.

At every *malloc()* call site, we insert code to set the corresponding base and bound. The `base` value is set to the pointer returned by *malloc()*. The `bound` value is set to either the pointer plus the size of the allocation (if the pointer is non-NULL) or to NULL (if the pointer is NULL) as shown in Figure 3.1(a). The bounds of global objects and stack allocated objects are known statically and the base and bound for a pointer pointing to a global variable/stack allocated object is set from the statically known size information as shown in Figure 3.1(b).

Spatial Safety Check Whenever a pointer is being dereferenced in the program, the pointer-based checking approach inserts a spatial dereference check for checking the bounds to detect spatial errors as shown in Figure 3.1(c). The spatial dereference check explicitly includes the size of the memory access to ensure that the entire access is in bounds (and not just the first byte). For example,

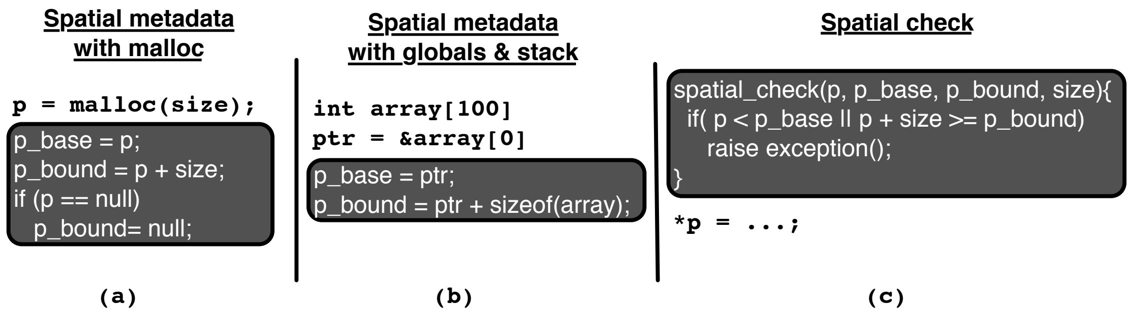


Figure 3.1: Metadata and checks for detecting spatial safety violations.

if a pointer to a single character is cast to be a pointer to an integer, dereferencing that pointer is a spatial violation.

3.4 Temporal Memory Safety Metadata and Checking

Temporal errors are consequence of accessing memory locations that have been deallocated. To detect temporal errors, our approach needs to track memory allocations and deallocations along with maintaining appropriate metadata with pointers. Memory allocations occur when (1) memory is explicitly allocated on the heap using a memory manager, (2) on the creation of stack frames on function entry, and (3) when the program initially allocates a global segment. Memory deallocations occur when memory is explicitly freed (*i.e.* using *free()*) or when a function returns. On every memory allocation, our approach assigns a unique identifier and the identifier is associated with the pointer pointing to the allocated memory. On memory deallocations, the identifier associated with the pointer to memory being deallocated is marked as invalid. On every memory access, the approach checks to ascertain if the identifier associated the pointer being dereferenced is still valid.

3.4.1 Lock and Key Metadata

Checking whether an identifier associated with the pointer is valid can be implemented by using a hashtable. However performing lookups into the hastable on every memory access can have significant overheads [96]. To meet our low overhead goals, inspired by prior proposals such as MSCC [131] and Patil & Fischer [103], we engineer the unique identifier into two sub-components to make the lookups fast: a *key* (a 64-bit unsigned integer) and a *lock* (a 64-bit address which

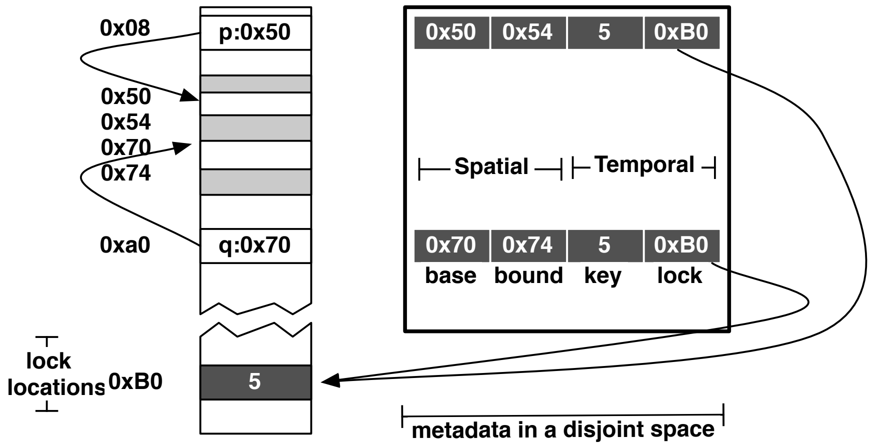


Figure 3.2: Metadata with each pointer: Bounds metadata and identifier (ID) metadata organized as key and lock where lock points to a lock location. The lock location pointed by the lock part of the ID has the same key as the key associated with the pointer. Here two pointers p and q point to different parts of the same object and hence have the same ID but different bounds.

points to a location in memory).[1] The memory location pointed by the lock is called the *lock location*. The lock provides an efficient mechanism for determining whether the memory allocated for the pointer has been deallocated. When memory is deallocated, we "change the lock" on the memory, preventing pointers with the now-stale key from accessing the memory (analogously to how a landlord might change the lock on a door after tenants move out). To accelerate checks, the system maintains the invariant that if the identifier is valid, the value stored in the lock location is equal to the key's value. Figure 3.2 shows the lock and key metadata maintained by each pointer to provide comprehensive memory safety.

In the subsections below, we describe the actions performed by our approach on memory allocations and deallocations on the heap and the stack. As global variables are never deallocated, we associate the constant identifier `GLOBAL_KEY` and the always-valid lock `GLOBAL_LOCK` with any pointer derived from a pointer to a global.

Allocations and Deallocations on the Heap When pointers point to memory allocated on the heap (*e.g.*, *malloc()* or `mmap()`), we insert additional code to: (1) associate a new unique key with the allocation pointer by incrementing the global key counter `next_key`, (2) obtain a new lock

[1] We have intentionally reversed the meaning of lock and key from original paper [103], because we find this terminology more intuitive—there can be multiple copies of the key for each lock location.

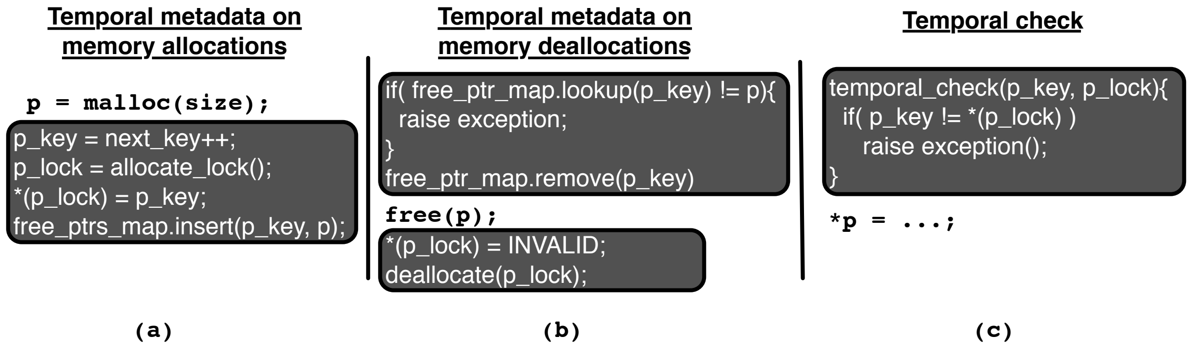


Figure 3.3: Metadata and checks for detecting temporal safety violations.

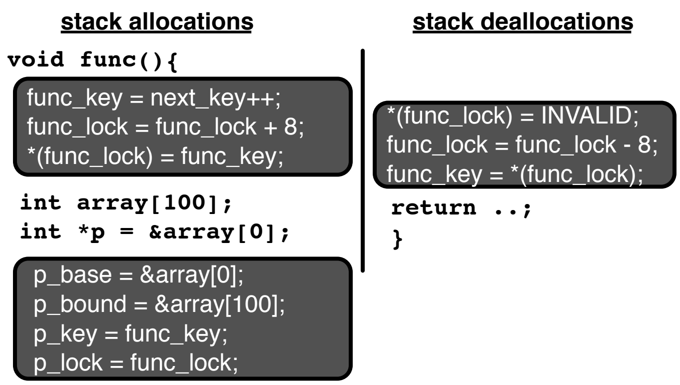


Figure 3.4: Temporal metadata management for the stack frame. On function entry, key and lock is allocated and the lock is deallocated on a return.

location, (3) write the key into the lock location, and (4) record that the pointer returned is free-able as shown in Figure 3.3(a).

For code that deallocates heap memory (*e.g.*, *free()* or `unmmap()`), we insert code to: (1) check for double-free and invalid-free by querying the free-able pointers map, removing the mapping if the free is allowed, (2) setting the lock's value to `INVALID_KEY`, and (3) deallocating the lock location as shown in Figure 3.3(b).

Stack Frame Allocation and Deallocation To detect temporal safety violation via dangling pointers to the call stack, a key and corresponding lock address is also associated with each stack frame. This key and lock address pair is given to any pointer derived from the stack pointer (and thus points to an object on the stack). Performing a stack allocation is much like calling *malloc()*, except that stack pointers are not free-able, so `free_ptrs_map` is unchanged. We manage the lock locations for the stack frames as a stack of lock locations for two reasons:(1) to increase locality in

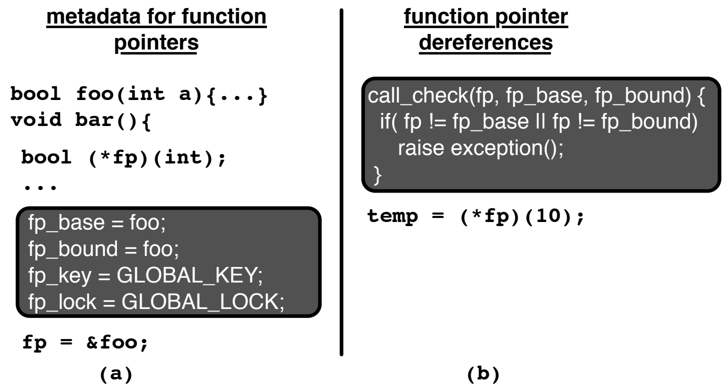


Figure 3.5: Metadata and checks for function pointers

the access of lock locations and (2) to simplify lock location allocation/deallocation. On function entry, a new key is allocated, a lock location is allocated by incrementing the stack lock location pointer. On a deallocation, the stack lock location pointer is decremented. On a function return, we also set the key associated with the stack frame to the value of the previous stack frame as shown in Figure 3.4.

3.4.2 Temporal Safety Checks

Once the lock and key metadata is maintained with pointers, our approach detects temporal safety violations by inserting a check of that metadata before every memory access as shown in Figure 3.3. The `temporal_check()` function checks if the key associated with the pointer is equal to the key at the lock location pointed by the lock associated with the pointer as shown in Figure 3.3. This check works because: (1) the key is written into the lock location at allocation time, (2) the contents of the lock location is changed upon deallocation, and (3) keys are unique and thus no subsequent allocation will ever reset the lock location to value that would cause a spurious match (even if the underlying memory or the lock locations are subsequently reused).

3.5 Control Flow Integrity Checks

The pointers in C programs can also point to functions. Such function pointers are later used to perform indirect function calls. Function pointers can be used to jump to arbitrary locations in memory. To ensure that the control flow integrity of the program is preserved, our approach assigns

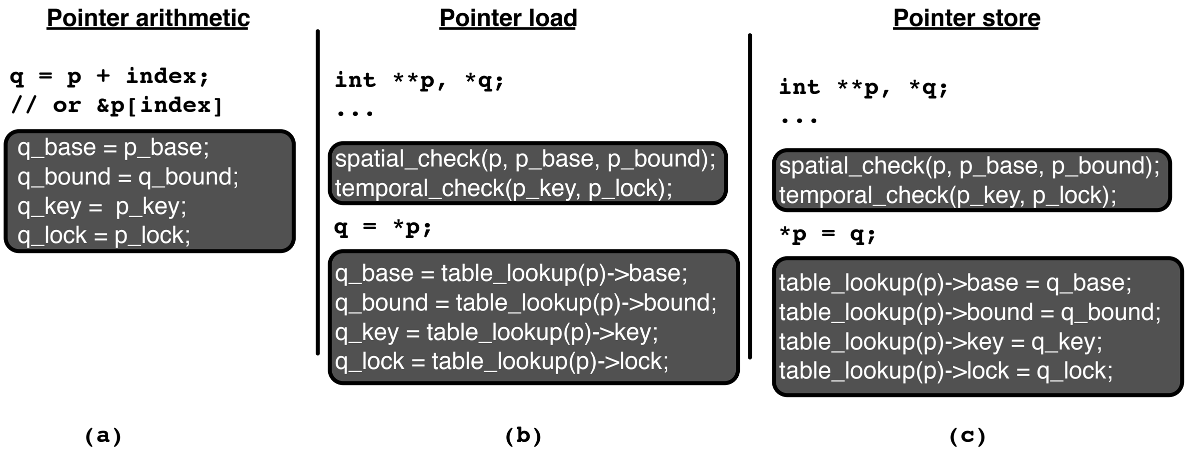


Figure 3.6: Pointer metadata propagation with pointer arithmetic and pointer loads/stores

metadata when such function pointers are created and checks all indirect function calls. On creating a function pointer using the address-of (&) operator on a function in the program, we set the base and bound of such a function pointer to be equal to the address of the function as shown in Figure 3.5(a). The key and lock metadata of function pointers is identical to that of global variables. This metadata with function pointers is propagated as with other pointer operations. When a function is called using a function pointer, the function pointer is checked to ensure that it is equal to its base and bound as shown in Figure 3.5(b). Such an encoding is not used by data objects (it would correspond to a zero-sized object), so our approach can check for this metadata when the program calls through a function pointer. This metadata encoding prevents data pointers or non-pointer data from being interpreted as a function pointer.

3.6 Propagation on Pointer Arithmetic and Assignment

To perform safety checks on a pointer dereference, we need to propagate metadata with pointer operations. When an expression contains pointer arithmetic (*e.g.*, `ptr+index`), array indexing (*e.g.*, `&(ptr[index])`), or pointer assignment (*e.g.*, `newptr = ptr;`), the resulting pointer inherits the metadata of the original pointer as shown in Figure 3.6. No checking is needed during pointer arithmetic because pointers are checked when dereferenced. As is required by C semantics, creating an out-of-bound pointer is allowed. We will detect the spatial violation whenever such a pointer is dereferenced. Array indexing in C is equivalent to pointer arithmetic, so we apply this

Shrinking bounds on structure field accesses

scalar sub-field

```
struct { ...; int num; ...} *n;
...
p = &(n->num);
```

```
p_base  = max(&(n->num), n_base);
p_bound = min(p_base + sizeof(n->num), n_bound)
```

array sub-field

```
struct { ...; int arr[5]; ...} *n;
...
p = &(n->arr[2]);
```

```
p_base  = max(&(n->arr), n_base);
p_bound = min(p_base + sizeof(n->arr), n_bound)
```

Figure 3.7: Optional shrinking of bounds for structure field accesses

same transformation to array indexing. Similarly, accesses to the fields of a structure are covered by the above transformations by conversion to separate pointer arithmetic and dereference operations.

3.7 Optional Narrowing of Pointer Bounds

The pointer-based approach adopted by this dissertation enables the ability to easily narrow the bounds of pointers, which in turn allows us to prevent internal object overflows. Shrinking of bounds can result in false violations for particularly pathological C idioms (discussed below), so we propose to shrinks pointer bounds only when explicitly instructed by the programmer to do so (*e.g.*, via a command-line flag when invoking the compiler).

When instructed to check for overflows within an object, we propose to shrink the bounds on a pointer when creating a pointer to a field of a *struct* (*e.g.*, when passing a pointer to an element of a *struct* to a function). In such cases, we narrow the pointer's bounds to include only the individual field rather than the entire object. The code in Figure 3.7 calculates the maximum base and minimum bound to ensure that such an operation will never expand the bounds of a pointer. Pointers to *struct* fields that are internal arrays (the size of which are always known statically) are handled similarly as shown the array subfields of Figure 3.7.

Although such narrowing of bounds may results in false positives, we have not encountered any false violations in any of our benchmarks. Yet, some legal C programs may rely on certain idioms that cause false violations when narrowing bounds. For example, a program that attempts to operate on three consecutive fields of the same type (*e.g.*, *x*, *y*, and *z* coordinates of a point) as a three-element array of coordinates by taking the address of *x* will cause a false violation. Another example of an idiom that can cause false violations comes from the Linux kernel's implementation of generic containers such as linked lists. Linux uses the ANSI C *offsetof()* macro to create a *container_of()* macro, which is used when creating a pointer to an enclosing container *struct* based only on a pointer to an internal *struct* [81]. Casts do not narrow bounds, so one idiom that will not cause false violations is casting a pointer to a *struct* to a *char** or *void**.

Another case in which we do not narrow bounds is when when creating a pointer to an element of an array. Although tightening the bounds is such cases may often match the programmer's intent, C programs occasionally use array element pointers to denote a sub-interval of an array. For example, a program might use *memset* to zero only a portion of an array using *memset(&arr[4], 0, size)* or use the *sort* function to sort a sub-array using *sort(&arr[4], &arr[10])*.

3.8 Disjoint Metadata

One of the key reasons that our pointer-based checking approach attains the aforementioned memory safety enforcement goals is due to the use of disjoint metadata. With disjoint metadata, our pointer-based checking avoids the memory layout changes and the resulting source incompatibilities and non-comprehensive detection that arises in the presence of arbitrary unsafe type casts as described in Section 2.3.4 of Chapter 2. The key challenge with disjoint metadata is in organizing it and providing the mapping functions to access it. In this section, we describe how the program accesses the disjoint metadata space and subsequently illustrate why this approach enables us to provide comprehensive detection of memory safety errors.

3.8.1 Mapping Pointers to their Metadata

Pointers are either resident in registers (when they are register allocated) or resident in memory. When they are resident in registers, the metadata can also be maintained in registers/temporaries and our approach maintains a mapping between the metadata and the pointers. However, when the pointer is resident in memory, the metadata is also resident in memory. When the pointer is loaded into a register/temporary by accessing memory, the metadata corresponding to the loaded pointer also needs to be accessed from the disjoint metadata space. Our approach uses a table data structure to map a pointer to its corresponding metadata.

Conceptually, we need to shadow the entire virtual address space to maintain metadata for pointers as a pointer can be resident in any address of the virtual address space. A concrete instantiation of the pointer-based checking approach with disjoint metadata can choose an appropriate table lookup structure (*e.g.*, hashtable, trie or even a linear array of bytes) depending on the performance and implementation tradeoffs. Our compiler instantiation uses a trie data structure that will be described in Chapter 4. Further, a hardware instantiation uses a linear array of bytes as the disjoint metadata space as it is easier to create and maintain address spaces with hardware support that will be described in Chapter 5.

Apart from the table lookup structure, a pointer based checking approach with disjoint metadata needs to figure out what it should use to index the table lookup data structure. To perform such lookups in the disjoint metadata space, we use the address of the pointer being loaded/stored to index into the table lookup structure to access the metadata space. Figure 3.6(b) illustrates the table lookups inserted to load the metadata. Similarly, when a pointer is being stored to memory, the metadata associated with the pointer is stored to memory using table lookups as shown in Figure 3.6(c).

In Figure 3.6(b), pointer q is loaded from memory by dereferencing pointer p. We perform the table lookup using the address of pointer q, which is p in our example. Using the address of the pointer is one of the important differences between our pointer-based approach with disjoint metadata and prior location based approaches (described in Chapter 2) that use disjoint metadata. Location-based approaches use the referent object (*i.e.*, what the pointer points to, which is q) to load the metadata in contrast to the address of the pointer (which is p).

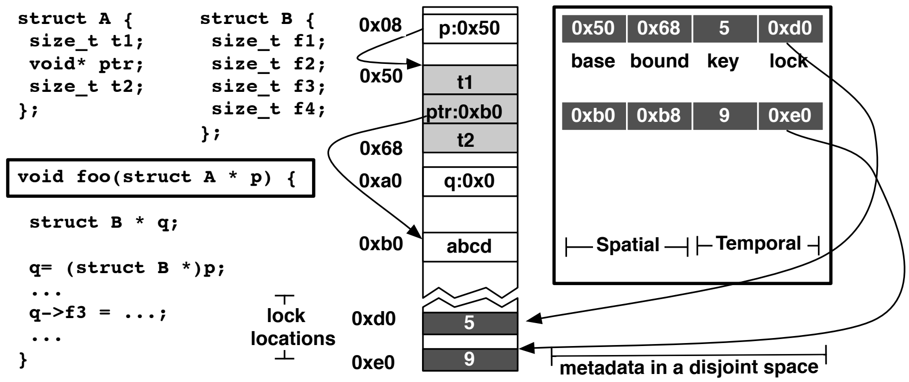
As our approach leaves the memory layout of the program intact with disjoint metadata, it avoids the problem of source incompatibilities and non-comprehensive detection. However, accessing the metadata when a pointer is loaded (stored) from (to) memory becomes expensive as it requires a mapping and translation operation. These expensive operations are required only when a pointer is loaded (stored) from (to) memory; loads and stores of non-pointer values are unaffected. Hence, a program which performs pointer arithmetic operations (including array accesses) and subsequently dereferences these pointers to load/store non-pointer values incur no disjoint metadata related overheads. On the other hand, linked data structures which load/store pointers to memory require accesses to the disjoint metadata space. Even though loads and stores of pointers are only a fraction of all memory operations, fast table lookups and updates are key to reducing overall overheads. Different organizations of the disjoint metadata have different performance characteristics and are discussed in detail in the concrete instantiations of this approach described in the subsequent chapters.

3.8.2 Comprehensive and Compatible Detection with Disjoint Metadata

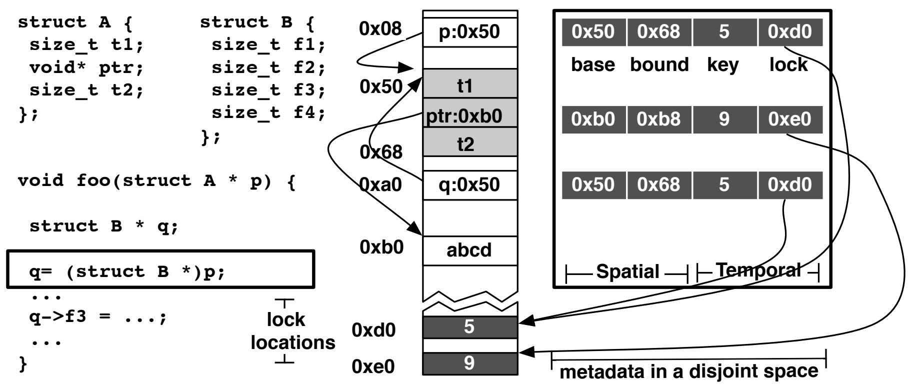
This section intuitively describes how disjoint metadata enables our approach to provide comprehensive detection of memory safety errors even in the presence of arbitrary type casts. A detailed formal proof of the memory safety guarantees of our approach was carried out in a collaborative setting for a rich subset of C and is described in our ISMM 2010 paper [93]. The invariants that enable us to enforce memory safety are: (1) the metadata is manipulated/accessed only by extra instrumentation added by our approach, (2) the metadata is never corrupted and the metadata accurately depicts the region of memory that a pointer can legally access (according to the C standard), and (3) all memory accesses are conceptually checked before a dereference.

The first invariant is satisfied from the way we have structured our instrumentation. The second invariant is satisfied always as there is no way to corrupt the metadata. Unsafe type casts were the primary cause of potential metadata corruption with fat pointer approaches. With disjoint metadata, any write using an arbitrarily type-casted pointer can overwrite only pointer values but not the metadata. When pointers involved in arbitrary casts are subsequently dereferenced, the pointer is checked with respect to its metadata. As the correctness checking uses the metadata to ascertain the validity of the memory access and the metadata is never corrupted, our approach ensures comprehensive detection of memory safety errors. Further programs do not need to be rewritten to avoid such casts. Hence, our approach works with legacy code without requiring modifications providing source compatibility.

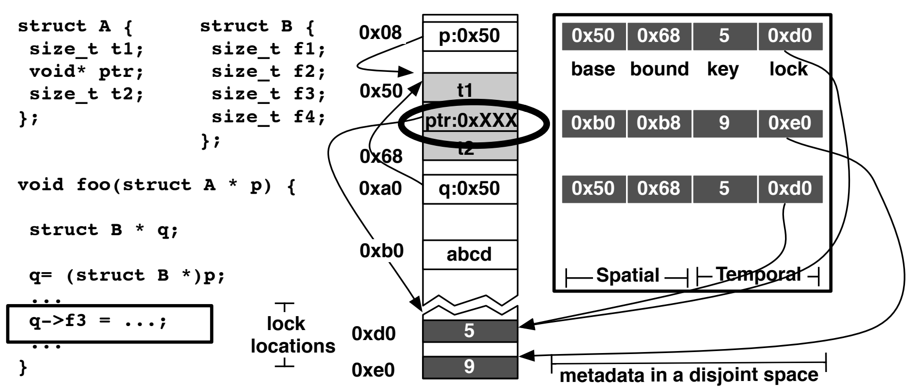
Figure 3.8 pictorially represents the operation of the disjoint metadata in the presence of arbitrary type casts. Figure 3.8(a) shows a program which has two structure types *A* and *B*. The function *foo* takes a pointer *p* as an argument that points to *struct A*, which is allocated and resident in memory as shown. The subfield *ptr1* of *struct A* is a pointer that points to some memory valid memory location. Pointers *p* and *ptr* subfield of *struct A* are resident in memory and have the appropriate metadata as shown in the disjoint metadata space. Figure 3.8(b) shows how the program creates a pointer *q* from pointer *p* by arbitrarily type-casting it to be a pointer of type *struct B*. The metadata of the pointer *q* is set to be equal to that pointer *p* in the disjoint metadata space assuming



(a) Pointer p points to structure A in memory. The subfield of A points to another location in memory.



(b) Pointer q points to the structure pointed by p considering it to be of type struct B.



(c) Pointer q writes a junk value into the ptr field. However the metadata is untouched and still consistent.

Figure 3.8: This figure illustrates how disjoint metadata protects the metadata. As a result, writes to memory location involved in arbitrary type casts can only modify pointer values (*ptr* field in the *struct A*) but not the metadata. When the pointer *ptr* is dereferenced, the dereference will not be allowed and the memory safety violation would be caught.

pointer *q* is resident in memory. Figure 3.8(c) shows how the program writes integers to memory locations using pointer *q*. As a result, *ptr* subfield could be overwritten with any arbitrary non-pointer value. However the metadata is not corrupted. When the program later tries to dereference *ptr* in memory, the dereference would be checked with respect to its metadata and memory safety errors would be detected.

Interfacing with the libraries was another problem with fat pointers as described in Chapter 2. Disjoint metadata also makes interfacing with libraries easier. Libraries that do not manipulate or change the pointers in the data structure work without the need for any marshaling as the program memory layout is unchanged. Libraries that manipulate pointers and expose such pointers to the program need either recompilation or wrappers to update the metadata. Although disjoint metadata makes interfacing with libraries without recompilation easier, we recommend recompiling libraries with our approach to attain comprehensive memory safety.

3.9 Modular Checking with Separate Compilation

Modern software is generally built with individual files compiled as separate modules and subsequently linked together. Supporting such separate compilation is essential to enable our approach to be applied to real world code. Our approach's instrumentation is purely local, operating on a function at a time. A function needs to interact with other functions only while passing and returning pointers. A detailed explanation of how we propagate metadata for pointer arguments and return values is described in Chapter 4. With local transformations, our approach enforces memory safety even with separate compilation and handles external libraries seamlessly in contrast to prior proposals that exploit whole program analysis [46, 94, 120]. The only constraint with our approach is that we need to compile all the modules/files with our approach. Otherwise, code compiled without our approach needs to interface with code compiled using our approach using wrappers that suitably updates the metadata to avoid false violations. In the absence of such wrappers, the program may experience false memory safety violations as we err on the side of conservatism (*i.e.*, safety first). We describe the wrappers for the compiler instantiation of this approach in Chapter 4.

3.10 Checking Modes With Disjoint Metadata

We experiment with two checking modes with our pointer-based checking approach with disjoint metadata to reduce the performance overheads. The default checking mode is the *full checking* mode, where the pointer-based checking approach performs a spatial and a temporal check conceptually before every memory access. The full checking mode is comprehensive in its detection of all memory safety errors but checking every memory operation can be expensive (experimentally evaluated in Chapter 4).

To mitigate performance overheads, we use a *store-only checking* mode that checks just store operations. Most security vulnerabilities exploiting a memory safety error use a store operation to inject code. Our experiments with security vulnerabilities reinforce the intuition that checking stores can prevent such security vulnerabilities in accordance with prior research [92, 133].

Although effective against memory safety errors and the concomitant security vulnerabilities that involve stores, the store-only checking mode does not detect errors that read from out-of-bound locations or invalid locations. As a result, store-only checking will not provide protection against information leaks. Store-only checking mode is purely a performance overhead reduction scheme in an effort to enable the deployment of pointer-based checking with real world code.

However, store-only checking mode does detect and prevent memory corruption involving stores to invalid locations. Such memory corruption resulting from memory safety errors involving stores to invalid locations is particularly insidious because they are harder to diagnose and the manifestation of the bug is often widely separated from the root cause location at which the memory corruption occurred. A important point to note is that the store-only mode reduces only the number of check operations. The number of metadata accesses and writes is generally unchanged because the same pointer metadata must be propagated to check subsequent store dereferences. However, an optimization could remove some of the metadata propagation instructions that feed only load dereference checks.

3.11 Usage Model

The pointer-based checking with disjoint metadata would likely first be deployed on an opt-in basis similar to the deployment scenario with Microsoft's page-based no-execute Data Execution Prevention (DEP) feature. Initially, DEP was not enabled by default as it can break some programs that use self-modifying code or JIT-based code generation. Similarly, our approach would not be enabled for programs that might violate our assumptions until after developers have explicitly tested their code with it. Like DEP, which is now enabled by default for most new software, we anticipate similar adoption for our approach.

3.12 Evaluation of Effectiveness in Detecting Errors

To validate our claim that pointer-based checking with disjoint metadata is effective in enforcing memory safety, we show that the approach detects known spatial safety and temporal safety violations. We use the prototype compiler instantiation that will be described in detail in Chapter 4 to perform these experiments. To test the effectiveness, we ran the prototype with our regression test suites, synthetic test suites [98], program extracts from real world attacks [129] and applications with known memory safety violations [87]. These test suites contained both versions of the program with and without memory safety violations. In our experimentation with these test suites, we not only detected known memory safety violations but also discovered new unknown memory safety violations.

3.12.1 Spatial Safety

To evaluate the effectiveness in detecting violations of spatial safety, we applied the compiler prototype to a suite of security violations [129], to versions of programs with well-documented security violations [87], to the SAFECode test suite from the experimental LLVM trunk, and to more than 2000 spatial safety violations from NIST Juliet Test Suite for C/C++ [98]. These suites include overflows on the stack, heap, and global segments to overwrite various return addresses, data pointers, function pointers, and longjmp buffers. Our approach detected all the spatial violations and prevented all the security vulnerabilities in these tests without any false positives.

To test the approach's effectiveness in detecting real world attacks, we use a test bed of buffer overflow attacks [129]. Table 3.1 lists the attacks based on the technique adopted, location of the overflow, and the attack target that is used to change the control flow. We detected all these errors without false violations. We also evaluated the approach's ability to detect spatial bugs using spatial errors from real programs obtained from the BugBench suite [87]: go, compress, gzip, and

Attack and Target	Detection	
	Store-only	Full
Buffer overflow on stack all the way to the target		
Return address	yes	yes
Old base pointer	yes	yes
Function ptr local variable	yes	yes
Function ptr parameter	yes	yes
Longjmp buffer local variable	yes	yes
Longjmp buffer function parameter	yes	yes
Buffer overflow on heap/BSS/data all the way to the target		
Function pointer	yes	yes
Longjmp buffer	yes	yes
Buffer overflow of a pointer on stack and then pointing to target		
Return address	yes	yes
Base pointer	yes	yes
Function pointer variable	yes	yes
Function pointer parameter	yes	yes
Longjmp buffer variable	yes	yes
Longjmp buffer function parameter	yes	yes
Buffer overflow of pointer on heap/BSS and then pointing to target		
Return address	yes	yes
Old base pointer	yes	yes
Function pointer	yes	yes
Longjmp buffer	yes	yes

Table 3.1: Various synthetic attacks proposed by Wilander *et al.* [129] and our approach's ability to detect them with full checking and store-only checking.

	Violation Detected?					
	Dissertation		Valgrind		GCC's	J&K
Benchmark	**Store-only**	**Full**	Memcheck	Ptrcheck	Mudflap	
go	no	yes	no	no	no	yes
compress	yes	yes	yes	yes	yes	yes
polymorph	yes	yes	no	yes	yes	yes
gzip	yes	yes	yes	yes	yes	yes

Table 3.2: Programs with overflows and the detection efficacy of our approach (store-only and full checking), Valgrind's memcheck [116], Valgrind's ptrcheck [95], GCC's Mudflap [51], and Jones and Kelly [75].

polymorph. These bugs are a mixture of one or more read or write overflows on the heap, stack, and globals. Our prototype detected these errors.

As a point of comparison, Table 3.2 also reports the efficacy of memcheck [116] and ptrcheck [95] from version 3.4.1 of Valgrind, Mudflap [51] from GCC 4.2.0, and the Jones and Kelly [75] modification to version 4.0.0 of GCC. Like this dissertation's approach, the Jones and Kelly version of GCC detected all violations. In contrast, both Valgrind and Mudflap detect some of the violations, but they also fail to detect violations that our approach detects. For example, Valgrind's memcheck tool does not detect overflows on the stack, leading to its failure to detect some of the bugs.

3.12.2 Temporal Safety

To evaluate the effectiveness in preventing use-after-free security exploits, we ran the 291 test cases for use-after-free vulnerabilities (CWE-416 and CWE-562) from the NIST Juliet Test Suite for C/C++ [98], which are modeled after various use-after-free errors reported in the wild. The prototype successfully detected and thwarted the attack in all the 291 test cases, and it did so without any false positives. We also evaluated our approach with tests from the SAFECode test suite from the experimental LLVM trunk. These tests included dangling pointers on the stack, and the heap. Our approach detected all these temporal safety violations without any false violations.

3.12.3 Previously Unknown Safety Errors

We discovered new previously unknown buffer overflow and use-after-free errors in the H.264 application of SPEC 2006 benchmark suite. The reference implementation of H.264 was reading a word beyond the bounds of an array (off by one error). We also discovered a previously unknown buffer

File: A.c

```
// global array of 100 ints
int array[100];

void foo() {
 int* p;
 ...
 p = array;

 p_base = array;
 p_bound = array+ sizeof(array);

}
```

File: B.c

```
// declared as a extern
extern int array[50];

void bar() {
 int* q;
 ...
 q = array;
 // sizeof(array) is 0
 q_base = array;
 q_bound = unbound();

}
```

Figure 3.9: Two files (A.c and B.c) use global array, which is defined in file A.c. In the absence of link-time optimizations, the bounds of the linker constant (array in file B.c) is unbounded.

---

overflow in the SPEC 2000 applications: 197.parser and 300.twolf demonstrating our approach's ability to detect memory safety violations.

3.13 Evaluation of Source Compatibility

One of the goals of pointer-based checking approach with disjoint-metadata proposed in this dissertation is to maintain source compatibility with existing C code. In this section, we describe our experience using the approach suggested in this dissertation with real world applications, and the issues that we had to address to handle real world C code.

Most real world applications use separate compilation where individual files are compiled separately into object files and later linked together. We observed that the build infrastructures are generally complicated to enable single module compilation without significant effort. Further, the libraries are also built using separate compilation and later linked with the applications. So supporting separate compilation is necessary to evaluate real world applications. Our approach supports separate compilation as described in Section 3.9. We were able to experiment with the pointer-based checking approach with disjoint-metadata for making both libraries and applications memory-safe.

There were two main issues that we had to address in our experimentation with real world applications. First, the size of the global variables declared in different translation units are not available until link time. Figure 3.9 illustrates this problem. The program has two files A.c and B.c. These files are compiled separately and instrumented using our approach. A global array of 100 integers is defined in file A.c and is used in both A.c and B.c. When this global array is assigned to a pointer in file B.c, our approach cannot use *sizeof(array)* to create the bounds as *sizeof()* returns zero for the extern global variables. The size of the array is not known until link time. In the absence of such information, the bounds assigned by our approach would be that of a zero sized array, and the program would experience false violations when such arrays are dereferenced. There are two alternatives to avoid such false violations: (1) to unbound such global arrays, and (2) to use link-time optimizations and complete the bounds information at link time. We use both approaches. In the common case where the link-time optimizations are not available, we use the first approach, and the second approach otherwise.

The second issue concerned function name changes. In our earlier paper on SoftBound [92], we changed the names of the functions that were transformed with our approach. However, changing the function names caused a few of the libraries to work incorrectly with default library build scripts. The utilities used to build libraries (ranlib, ar and others) required that the functions names be unchanged. Thus, we leave the function names unchanged with our approach.

To enable evaluation with existing libraries, we have also provided wrappers that update the metadata appropriately to interact with the instrumented code. We have run a wide range of applications, compiled libraries, and benchmark suites without source code modifications with our approach. We have successfully experimented with OpenSSL, SQLite, tar, flex and many other utilities. Apart from these utilities, our prototype also successfully transformed the benchmarks from the SPEC suite used in the performance evaluation. In total, these benchmarks and utilities are approximately one million total lines of code, all of which were successfully transformed, further supporting our source code compatibility claim.

3.14 Summary

This chapter presented our approach for enforcing memory safety using a pointer-based checking with disjoint metadata. The use of disjoint metadata is the primary reason that makes pointer-based checking attractive again. With disjoint metadata, the memory layout of the program is unchanged. A purely local transformation with disjoint metadata can enforce comprehensive memory safety even in the presence of casts as metadata can never be corrupted. Local transformations combined with disjoint metadata also enable separate compilation. Hence, memory safe libraries can be built with our approach providing high source compatibility for existing programs. We highlighted the basic approach without focusing on the performance overheads in this chapter. Subsequent chapters describe the concrete instantiations of this approach within the compiler (Chapter 4), within hardware (Chapter 5) and with hardware-assisted compiler instrumentation (Chapter 6) addressing the performance overheads thereby making it attractive for practical deployment.

Chapter 4

Compiler Instrumentation for Pointer-Based Memory Safety

This chapter describes the implementation of the pointer-based approach with disjoint metadata within the compiler. The goal of this compiler-based instrumentation is to provide a low overhead implementation leveraging the information available to the compiler. The compiler is a natural choice for performing such an instrumentation as (1) it is already part of every tool chain, (2) it maintains significant information about the program to perform semantics preserving optimizations, and (3) a memory safety instrumentation can be performed on optimized code. As we use *Low-Level Virtual Machine* (LLVM) [83] as our compiler, we initially provide background on the LLVM compiler and its intermediate representation in Section 4.1. Subsequently we describe the details of the memory safety instrumentation on the LLVM IR in Section 4.2. We describe the optimizations to the compiler instrumentation in Section 4.3. We evaluate the performance overheads of a compiler-based memory safety transformation in Section 4.4.

4.1 Background on LLVM

The *Low-Level Virtual Machine* (LLVM) is a platform-independent compiler that was originally developed by Lattner *et al.* to study optimizations and modern compilation techniques [83]. The LLVM project has now blossomed into a robust, industrial-strength and open-source compilation platform that competes with GNU C Compiler (GCC) in terms of compilation speed and the performance of the generated code. As a consequence, the LLVM has been widely used in both industry and academia, and it has a growing and very active user community.

The LLVM compiler is structured as a translation from a high-level source language to the LLVM Intermediate Representation (IR). The LLVM compilation framework provides a large (and user-extensible) suite of IR to IR translations, which provide many advanced optimizations, program transformations, and analyses. The resulting LLVM IR can then be lowered to a variety of target architectures, including x86, PowerPC, Arm, *etc.*, or JIT-compiled as desired. The standard LLVM suite provides C and C++ front-ends, along with ports for many other source languages.

4.1.1 Structure of the LLVM IR

The LLVM IR is one of the core components of the LLVM compiler. The LLVM IR is a typed, static single assignment (SSA) language that is a suitable representation for expressing many compiler transformations and optimizations. This section describes the syntax of the LLVM IR and the core features that are required to understand the compiler instrumentation for memory safety. Figure 4.1 shows the abstract syntax for the subset of the LLVM IR. The metavariable id ranges over LLVM identifiers, written `%X`, `%T`, `%a`, `%b`, *etc.*, which are used to name local types and temporary variables, and `@a`, `@b`, `@main`, *etc.*, which name global values and functions. Each source file is a module mod that includes data layout information $layout$ (which defines sizes and alignments for types; see below), named types, and a list of $prod$s that can be function declarations, function definitions, and global variables.

Every LLVM expression has a type, which can easily be determined from type annotations, that provides sufficient information to check an LLVM program for type compatibility. The LLVM IR is not a type-safe language, however, because its type system allows arbitrary casts, calling functions with incorrect signatures, accessing invalid memory, *etc.* The LLVM type system ensures only that the size of a runtime value in a well-formed program is compatible with the type of the value.

Types typ include arbitrary bit-width integers $\mathbf{i}8$, $\mathbf{i}16$, $\mathbf{i}32$, *etc.*, or, more generally, $\mathbf{i}sz$ where sz is a natural number. Types also include $\mathbf{float}$, **void**, pointers $typ*$, arrays $[\,sz \times typ\,]$ that have a statically-known size sz. Anonymous structure types $\{\,\overline{typ_j}^{\,j}\,\}$ contain a list of types. Functions $typ\,\overline{typ_j}^{\,j}$ have a return type, and a list of argument types. Here, $\overline{typ_j}^{\,j}$ denotes a list of typ components; we use similar notation for other lists throughout the thesis. Finally, types can be named

$$
\begin{array}{llcl}
\text{Modules} & mod, P & ::= & \overline{layout}\,\overline{namedt}\,\overline{prod} \\
\text{Layouts} & layout & ::= & \mathbf{bigendian} \mid \mathbf{littleendian} \mid \mathbf{ptr}\ sz\ align_0\ align_1 \\
 & & \mid & \mathbf{int}\ sz\ align_0\ align_1 \mid \mathbf{float}\ sz\ align_0\ align_1 \\
 & & \mid & \mathbf{aggr}\ sz\ align_0\ align_1 \mid \mathbf{stack}\ sz\ align_0\ align_1 \\
\text{Products} & prod & ::= & id = \mathbf{global}\ typ\ const\ align \mid \mathbf{define}\ typ\ id(\overline{arg})\{\overline{b}\} \\
 & & \mid & \mathbf{declare}\ typ\ id(\overline{arg}) \\
\text{Floats} & fp & ::= & \mathbf{float} \mid \mathbf{double} \\
\text{Types} & typ & ::= & \mathbf{i}sz \mid fp \mid \mathbf{void} \mid typ* \mid [\,sz \times typ\,] \mid \{\,\overline{typ_j}^{\,j}\,\} \mid typ\,\overline{typ_j}^{\,j} \mid id \\
\text{Values} & val & ::= & id \mid cnst \\
\text{Binops} & bop & ::= & \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{udiv} \mid \mathbf{sdiv} \mid \mathbf{urem} \mid \mathbf{srem} \mid \mathbf{shl} \mid \mathbf{lshr} \\
 & & \mid & \mathbf{ashr} \mid \mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \\
\text{Float ops} & fbop & ::= & \mathbf{fadd} \mid \mathbf{fsub} \mid \mathbf{fmul} \mid \mathbf{fdiv} \mid \mathbf{frem} \\
\text{Extension} & eop & ::= & \mathbf{zext} \mid \mathbf{sext} \mid \mathbf{fpext} \\
\text{Cast op} & cop & ::= & \mathbf{fptoui} \mid \mathbf{ptrtoint} \mid \mathbf{inttoptr} \mid \mathbf{bitcast} \\
\text{Trunc op} & trop & ::= & \mathbf{trunc}_{int} \mid \mathbf{trunc}_{fp} \\
\text{Constants} & cnst & ::= & \mathbf{i}sz\ Int \mid fp\ Float \mid typ * id \mid (typ*)\,\mathbf{null} \mid typ\ \mathbf{zeroinitializer} \\
 & & \mid & typ[\,\overline{cnst_j}^{\,j}\,] \mid \{\,\overline{cnst_j}^{\,j}\,\} \mid typ\ \mathbf{undef} \mid bop\ cnst_1\ cnst_2 \\
 & & \mid & fbop\ cnst_1\ cnst_2 \mid trop\ cnst\ \mathbf{to}\ typ \mid eop\ cnst\ \mathbf{to}\ typ \mid cop\ cnst\ \mathbf{to}\ typ \\
 & & \mid & \mathbf{getelementptr}\ cnst\ \overline{cst_j}^{\,j} \mid \mathbf{select}\ cnst_0\ cnst_1\ cnst_2 \\
 & & \mid & \mathbf{icmp}\ cond\ cnst_1\ cnst_2 \mid \mathbf{fcmp}\ fcond\ cnst_1\ cnst_2 \\
 & & \mid & \mathbf{extractvalue}\ cnst\ \overline{cnst_j}^{\,j} \mid \mathbf{insertvalue}\ cnst\ cnst'\ \overline{cnst_j}^{\,j} \\
\text{Blocks} & b & ::= & l\,\overline{\phi}\,\overline{c}\ tmn \\
\phi\ \text{nodes} & \phi & ::= & id = \mathbf{phi}\ typ\ \overline{[val_j, l_j]}^{\,j} \\
\text{Tmns} & tmn & ::= & \mathbf{br}\ val\ l_1\ l_2 \mid \mathbf{br}\ l \mid \mathbf{ret}\ typ\ val \mid \mathbf{ret\ void} \mid \mathbf{unreachable} \\
\text{Commands} & c & ::= & id = bop(\,\mathbf{int}\ sz)val_1\ val_2 \mid id = fbop\ fp\ val_1\ val_2 \\
 & & \mid & id = \mathbf{load}\,(typ*)val_1\ align \mid \mathbf{store}\ typ\ val_1\ val_2\ align \\
 & & \mid & id = \mathbf{alloca}\ typ\ val\ align \mid id = trop\ typ_1\ val\ \mathbf{to}\ typ_2 \\
 & & \mid & id = eop\ typ_1\ val\ \mathbf{to}\ typ_2 \mid id = cop\ typ_1\ val\ \mathbf{to}\ typ_2 \\
 & & \mid & id = \mathbf{icmp}\ cond\ typ\ val_1\ val_2 \mid id = \mathbf{select}\ val_0\ typ\ val_1\ val_2 \\
 & & \mid & id = \mathbf{fcmp}\ fcond\ fp\ val_1\ val_2 \mid option\ id = \mathbf{call}\ typ_0\ val_0\ \overline{param} \\
 & & \mid & id = \mathbf{getelementptr}\,(\,typ*\,)\ val\ \overline{val_j}^{\,j}
\end{array}
$$

Figure 4.1: Syntax for the LLVM IR.

by identifiers id. The sizes and alignments for types, and endianness are defined in $layout$. For example. **int** $sz\ align_0\ align_1$ dictates that values with type $\mathbf{i}sz$ are $align_0$-byte aligned when they are within an aggregate and when used as an argument, and $align_1$-byte aligned when emitted as a global.

The two key features of the LLVM IR that we leverage for the memory safety instrumentation are (1) the presence of type information in the IR to identify pointers and (2) SSA representation keeping the instrumentation simple by performing most of the work on temporaries.

4.1.2 Operations in the LLVM IR

Operations in the LLVM IR compute with values val, which are either identifiers id naming temporaries, or constants $cnst$ computed from statically-known data, using the compile-time analogs of the commands described below. Constants include base values (*i.e.* integers or floats of a given bit width), and zero-values of a given type, as well as structures and arrays built from other constants.

All code in the LLVM IR resides in top-level functions, whose bodies are composed of block bs. As in classic compiler representations, a basic block consists of a labeled entry point l, a series of ϕ nodes, a list of commands, and a terminator instruction. As is usual in SSA representations, the ϕ nodes join together values from a list of predecessor blocks of the control-flow graph—each ϕ node takes a list of (value, label) pairs that indicates the value chosen when control transfers from a predecessor block with the associated label. Block terminators (**br** and **ret**) branch to another block or return (possibly with a value) from the current function. Terminators also include the **unreachable** marker, indicating that control should never reach that point in the program.

The core of the LLVM instruction set is its *commands* (c), which include the usual suite of binary arithmetic operations (*bop*—e.g., **add**, **lshr**, etc.), memory accessors (**load**, **store**), stack allocation (**alloca**), conversion operations among integers, floats and pointers (*eop*, $trop$, and *cop*), comparison over integers (**icmp** and **select**), and calls (**call**). Note that a call site is allowed to ignore the return value of a function call. The LLVM IR also provides aggregate data operations (**extractvalue** and **insertvalue**) for projecting and updating the elements of structures and arrays. Finally, **getelementptr** computes pointer offsets into structured datatypes based on their types; it provides a platform- and layout-independent way of performing array indexing, struct field access, and pointer arithmetic.

Figure 4.2 and Figure 4.3 shows the C code and the corresponding LLVM IR code for constructing a simple circular linked list and traversing it for a few iterations respectively. The example illustrates the type information in the IR, SSA features in the IR, the use of getelementptr, phi nodes in the IR. We will use this example to illustrate the memory safety instrumentation over the IR.

C program for Circular Linked List

```
struct node_t {
  size_t value;
  struct node_t* next;
};
typedef struct node_t node;

int main(int argc, char** argv)
{
  node* fptr = malloc(sizeof(node));
  node* ptr = fptr;
  size_t i;

  fptr->value = 0;
  fptr->next = NULL;

  // init
  for (i=0; i<SIZE; i++) {
    node* new_ptr = malloc(sizeof(node));
    new_ptr->value = i;
    new_ptr->next = ptr;
    ptr = new_ptr;
  }
 // make it a circular linked list
  fptr->next = ptr;
  return 0;

}
```

Figure 4.2: A circular linked list example in C.

4.2 Memory Safety Instrumentation on the LLVM IR

The primary goal of performing the memory safety instrumentation within the compiler is to reduce the performance overhead by instrumenting optimized code. Beyond the performance overhead objective, the other goal is to keep the instrumentation simple so that it can be used with separate compilation for real world programs without requiring complicated whole program analyses as with prior approaches [44, 94]. Further, the simplicity of our instrumentation has enabled the design of a completely verified memory safety instrumentation within a proof assistant in a related but different project [137]. We use the name *SoftBoundCETS* for the compiler-based memory safety instrumentation described in this chapter as it builds on top of our prior work *SoftBound* (providing spatial safety) [92] and *CETS* (providing temporal safety) [93].

LLVM IR code for Circular Linked List

```
define i32 @main(i32 %argc, i8** %argv) nounwind uwtable {
entry:
  %call = call noalias i8* @malloc(i64 16) nounwind
  %0 = bitcast i8* %call to %struct.node_t*
  %value = getelementptr inbounds %struct.node_t* %0, i32 0, i32 0
  store i64 0, i64* %value, align 8
  %next = getelementptr inbounds %struct.node_t* %0, i32 0, i32 1
  store %struct.node_t* null, %struct.node_t** %next, align 8
  br label %for.cond

for.cond:
  %ptr.0 = phi %struct.node_t* [ %0, %entry ], [ %1, %for.inc ]
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i64 %i.0, 128
  br i1 %cmp, label %for.body, label %for.end


for.body:
  %call1 = call noalias i8* @malloc(i64 16) nounwind
  %1 = bitcast i8* %call1 to %struct.node_t*
  %value2 = getelementptr inbounds %struct.node_t* %1, i32 0, i32 0
  store i64 %i.0, i64* %value2, align 8
  %next3 = getelementptr inbounds %struct.node_t* %1, i32 0, i32 1
  store %struct.node_t* %ptr.0, %struct.node_t** %next3, align 8
  br label %for.inc

for.inc:
  %inc = add i64 %i.0, 1
  br label %for.cond

for.end:
  %next4 = getelementptr inbounds %struct.node_t* %0, i32 0, i32 1
  store %struct.node_t* %ptr.0, %struct.node_t** %next4, align 8
  ret i32 0
}
```

Figure 4.3: A circular linked list example LLVM IR.

Beyond the details described in the basic approach in Chapter 3, the compiler-based instrumentation needs to make design choices on three main dimensions: (1) how to propagate metadata for pointer arguments and returns in function calls, (2) how to organize the metadata space, and (3) how to add instrumentation code. We address these design issues in the following sections.

4.2.1 Metadata Propagation for Pointer Parameters and Returns

When pointers are passed as arguments or returned from functions, their metadata must also travel with them. If all pointer arguments were passed and returned on the stack (*i.e.*, via memory and not registers), the table-lookup described in Chapter 3 for handling in-memory metadata would suffice. However, the function calling conventions of most ISAs specify that function arguments are generally passed in registers.

One of the design decisions we had to make while designing the compiler instrumentation was to design a mechanism to propagate metadata for pointers when they are either passed as arguments to a function or when they are returned by a function. This turned out to be an important and tricky design decision to enable our compiler transformation to work with real world code. In our PLDI 2009 paper on SoftBound [92] and ISMM 2010 paper on CETS [93], we adopted procedure cloning [36] to transform every function declaration and function call site to include additional arguments for base, bound, key and lock. For each pointer argument, this metadata was added to the end of the list of function's arguments. Functions that return a pointer were changed to return a five-element structure by value that contained the pointer, its base, bound, key and lock. As part of that transformation, the function name was appended with a instrumentation specific identifier, specifying that the function has been transformed.

Although the above described approach where we changed function signatures worked as expected for most functions, we faced two problems. First, engineering it to work was extremely cumbersome as we had to rewrite significant portion of the LLVM IR code especially in the presence of indirect function calls and use of function pointers. Apart from rewriting a large amount of code, we had to introduce a large number of new types. For example, a structure type with a function pointer now required to be rewritten as a new type that had a new function pointer type whose prototype included extra arguments to the pass the metadata. A combination of bugs in tedious IR rewriting and aggressive optimizer which exploited type-unsafe casts resulted in the transformation being less robust with real world code.

Second, padding extra arguments at the end of the functions fails to detect mismatches in the number and position of pointer arguments at the call-site and the called function. Further, function pointers involved in indirect function calls could be cast from function pointers of incompatible types. As a result, calling a function with arbitrary values through a function pointer can manufacture improper metadata. Further, handling variable argument function is not straightforward with such an approach.

Metadata Propagation with Shadow Stack for Parameters and Return Values To address the above issues while being independent of the target ISA and obeying system's calling convention, we propose the use of a shadow stack for propagating metadata for pointer arguments and returns. The shadow stack is organized as a stack in a separate disjoint space in the program's virtual address space. It mirrors the call stack of the executing program. Unlike the call stack that holds the activation record, the shadow stack holds the metadata for the pointer arguments and return values for the frames in the call stack.

At the call site, the transformation adds extra instrumentation to push the number of arguments and metadata for each argument onto the shadow stack. For non-pointer arguments, the transformation pushes invalid metadata. On a function entry, the transformation inserts code to read the metadata for the pointer argument into a temporary and maintains the mapping between the pointer and the metadata. Further on a function exit, the callee pushes either the appropriate metadata when the function returns a pointer or invalid metadata in the case of non-pointer return values.

Figure 4.4 illustrates the load/store operations from/to the shadow both in the caller and the callee. Figure 4.4(a) depicts the shadow stack setup by the instrumented code before calling the function *foo*, which takes a pointer as an argument and returns a pointer. The shadow stack has reserved metadata space for the return values irrespective of whether the function returns a pointer or not. The caller initializes the return metadata to be invalid in the newly setup shadow stack frame. Figure 4.4(b) illustrates the retrieval of the metadata on function entry by the callee. Figure 4.4(c) shows the store operations performed by the callee to store the metadata of the returned pointer. Figure 4.4(d) shows the load operations performed by the caller to retrieve the metadata for the returned pointer. The shadow stack handlers that load the metadata take argument number as a

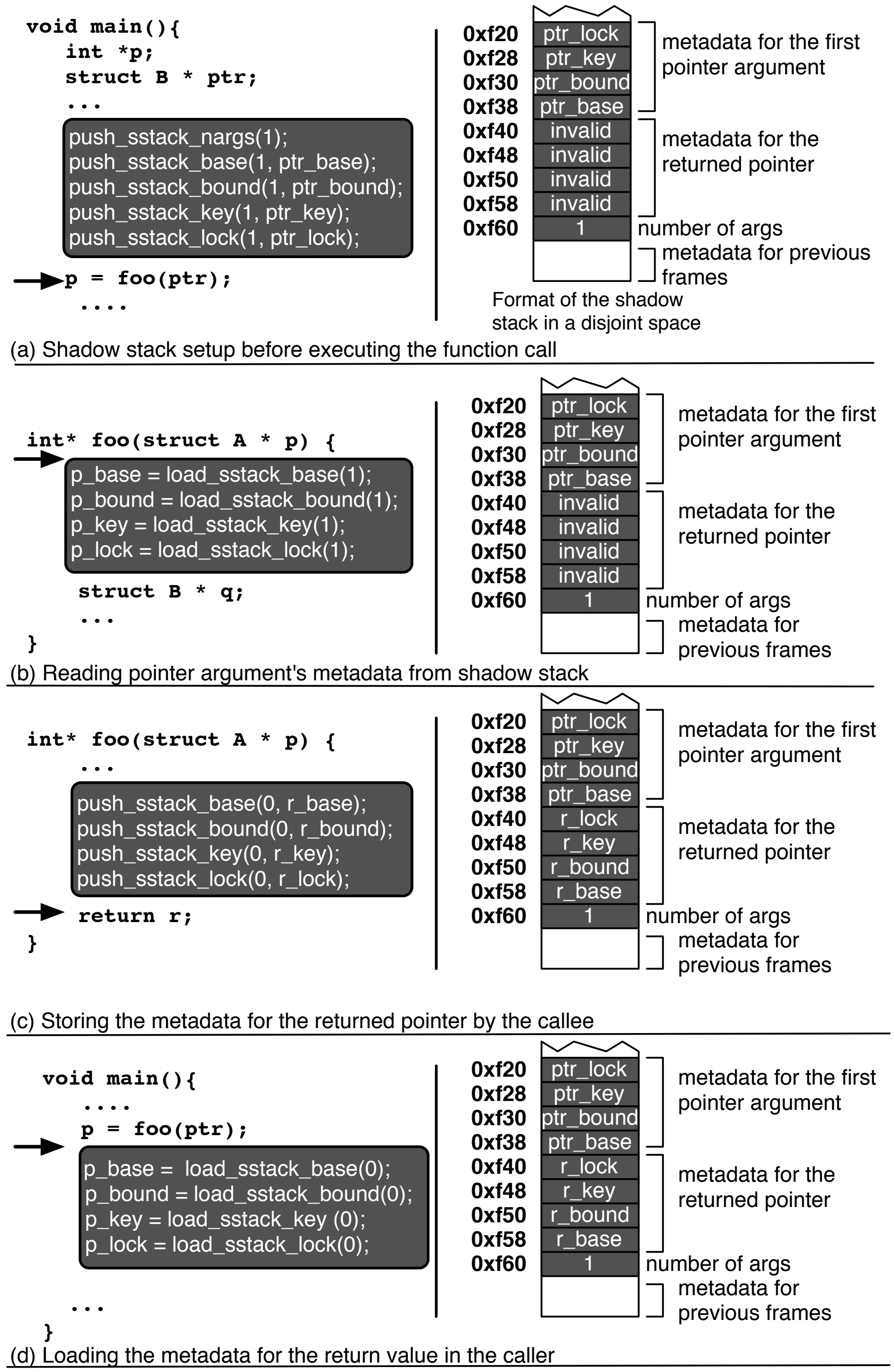


Figure 4.4: This figure illustrates the stores/loads to/from the shadow stack in the caller and the callee.

parameter. Similarly shadow stack handlers that store the metadata use the argument number and the metadata as the parameters as shown in the Figure 4.4.

In summary, the shadow stack provides a mechanism for dynamic typing between the arguments pushed at the call site and the arguments retrieved by the callee. The shadow stack ensures that the callee never coerces the program to treat an integer pushed by the caller in the call stack as a pointer and subsequently dereference it. An exception is triggered only when such pointers are dereferenced but not when they are created in accordance with the C specification. When the callee attempts to retrieve the metadata for the particular argument, it receives the metadata pushed by the caller (which would be either valid metadata when the caller pushes a pointer on the call stack or invalid metadata otherwise). The shadow stack handlers also ensure that the callee does not attempt to access arbitrary arguments on the call stack by checking the argument number of the parameter for which metadata is being retrieved with the number of arguments pushed by the caller. Thus, the shadow stack matches the callee view of the call stack with the caller's view with respect to pointer and non-pointer parameters and returns. The following section demonstrates how this mechanism provides safety even with variable argument functions.

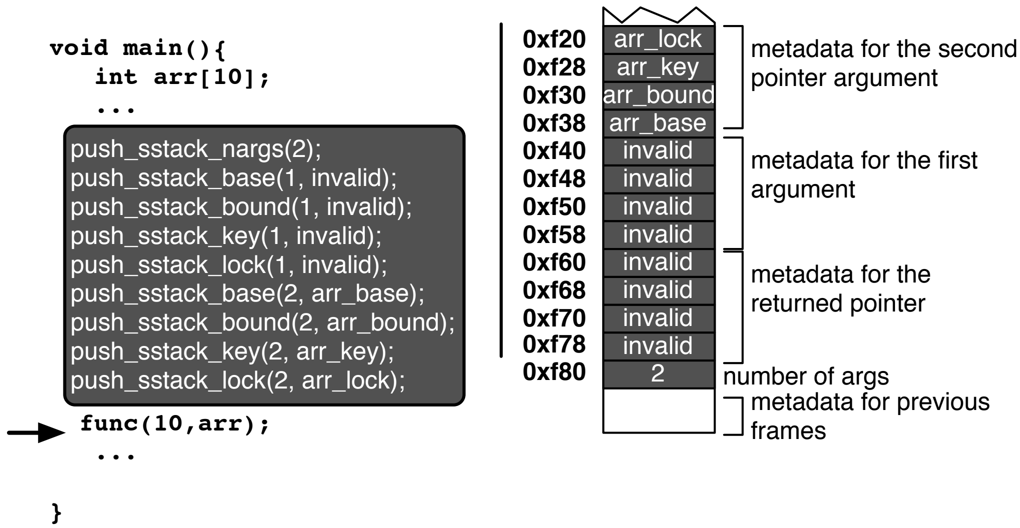
Handling Variable Argument Functions with Shadow Stack

Variable argument functions are common in C. The function prototype does not specify the number of arguments for these functions. The arguments passed to such a function at a call-site are retrieved using *va_list* handlers such as *va_start*, *va_arg*, and *va_end*. Variable argument functions are a common source of errors because unbounded number of arguments can be potentially read from the call stack. Such errors can be prevented using the shadow stack. The shadow stack provides accurate information about the number of arguments pushed by the call-site. Further, shadow stack also provides a mechanism to pass metadata for the arguments.

Figure 4.5(a) illustrates a simple variable argument function definition and such a function being called by another function. Figure 4.5(b) shows how the shadow stack is setup at the call-site. Figure 4.6(a) shows the checks performed whenever an argument is retrieved using the *va_arg* handler. In Figure 4.6(a), a pointer is being retrieved using the *va_arg* handler. Hence our transformation has inserted code to retrieve the metadata from the shadow stack. Figure 4.6(b) shows how the checks

```
void main(){                    void func(int num, ...){
   int arr[10];                    int *p;
   ...                             va_list args;
                                   va_start(args, num);
   func(10,arr);
                                   p = va_arg(args, int *);
   ...
                                   ...
}                               }
```

(a) Variable argument function with pointer as an argument



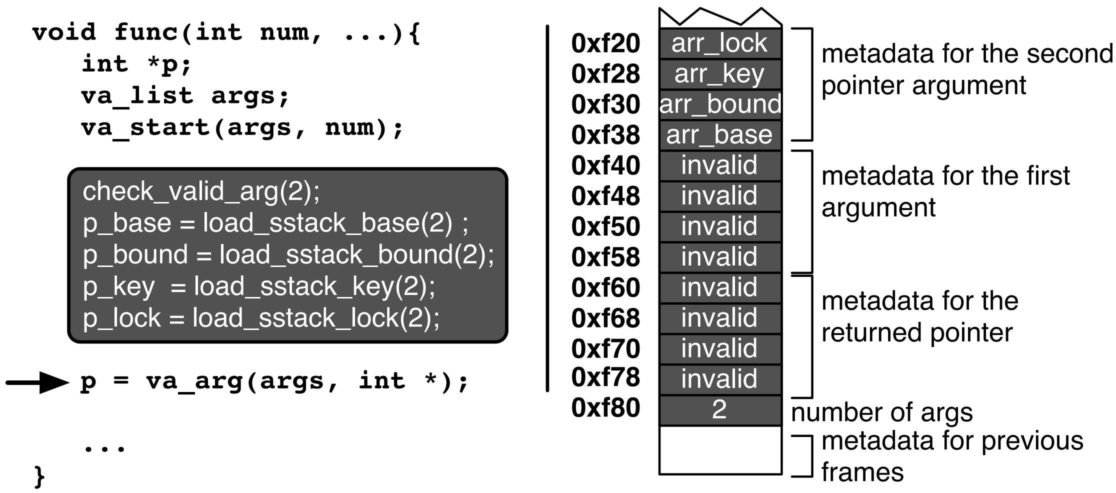
(b) Main function that calls the variable argument function sets up the shadow stack

Figure 4.5: This figure illustrates a simple function that takes a variable number of arguments in (a) and illustrates how the caller sets up the shadow stack for the variable argument function in (b).

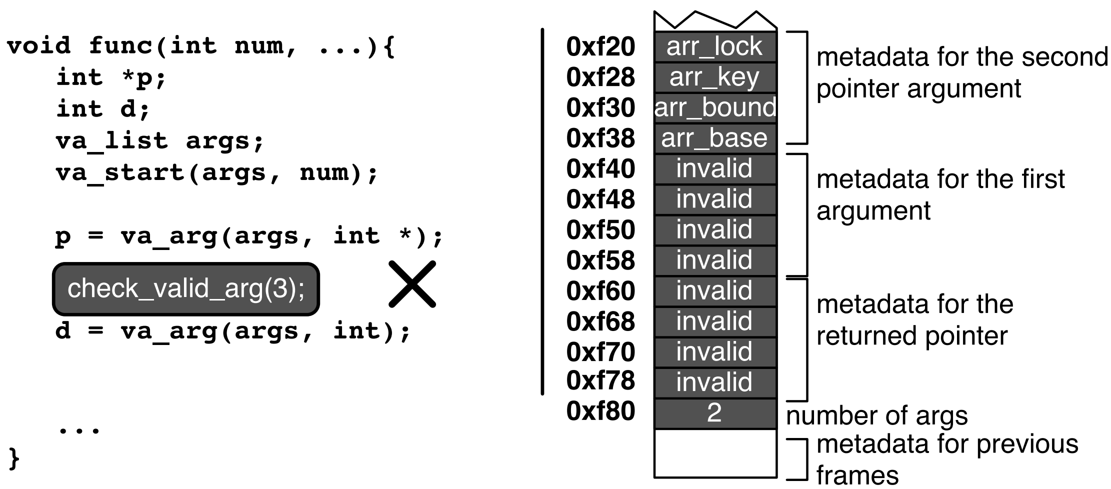
inserted by our transformation signal an exception when the function attempts to read arbitrary number of values from the stack.

4.2.2 Metadata Organization

We described the metadata facility abstractly using table lookups in Chapter 3. This section describes the organization of the metadata facility for the compiler transformation. In our prior work SoftBound [92] on providing spatial safety, we experimented with two different organizations of the metadata space: a hashtable organization and a shadow space organized as a linear array of bytes. However, memory overhead was an important issue that we had to address with the hashtable and the shadow space organizations. A pointer can be resident be in any address in the virtual address



(a) Callee, which is a variable argument function, retrieves the metadata from the shadow stack for the pointer argument



(b) An example where a variable argument function tries to load 3 arguments from the stack when the callsite pushed just two arguments. This is prevented by the check performed before loading each argument as indicated in the figure by X.

Figure 4.6: This figure illustrates how the callee retrieves the arguments in a variable argument function and checks that occur using the shadow stack in (a) and how the mismatch in the number of arguments pushed by the caller and callee is caught using the shadow stack in (b).

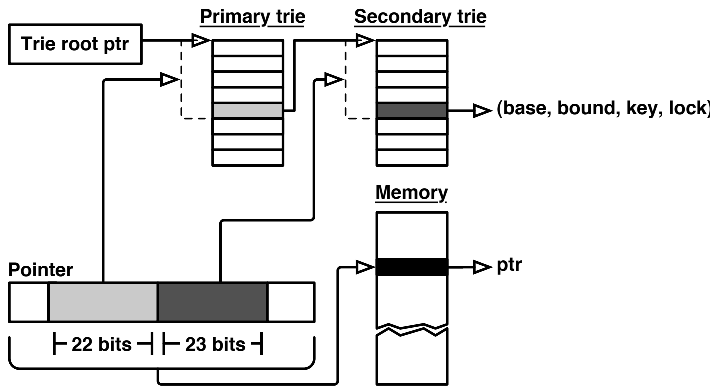


Figure 4.7: Organization of the trie for pointer metadata

space. Hence, we had to shadow the entire virtual address space. Shadowing the metadata space using a linear byte array based shadow space was not feasible as our metadata (32 bytes per word) is larger than word size and the current operating systems do not allow us to *mmap* regions larger than the size of the virtual memory plus the swap space. On the other hand, such shadowing with the hashtable required that we sized the hash table reasonably large and resize it on frequent collisions. We had to find an appropriate balance between the memory overhead incurred and the frequency of the resizes to determine the resizing granularity. Frequent resizes would increase the performance overhead. On the other hand, increasing the memory overhead significantly would prevent a program from running. Hence, to avoid such problems and incur memory overhead on demand, we organize the metadata space as a trie for our compiler transformation.

We implement the disjoint metadata facility using a two-level *lookup trie* data structure, which provides the ability to shadow every word in memory efficiently [96]. A trie is a mapping structure much like a page table in which each level of the trie is accessed using a set of bits from the index being accessed (see Figure 4.7). The current compiler transformation prototype assumes a 48-bit virtual address space and that pointers are word-aligned, for a total of 45 bits to index the trie. The prototype uses 2^{22} first level entries and 2^{23} second level entries. Each second-level entry contains the bounds metadata and the identifier metadata (256 bits total). Each metadata load and store includes a trie lookup. We allocate the first-level trie entries when a metadata store accesses the entry. The C code for the metadata load and store is shown in Figure 4.8. Each metadata load with a trie lookup is approximately twelve x86 instructions. It involves two loads (a load to retrieve the global base of the first level trie plus a load from the first level trie entry) and six bitmask and shift

C code for Metadata Load

```
/* struct metadata_entry_t {
    void* base;
    void* bound;
    size_t key;
   void* lock;
};
*/

struct metadata_entry_t
 sbcets_metadata_load(void* addr_of_ptr){

   struct metadata_entry_t* meta_entry;
   struct metadata_entry_t* secondary_table;
   size_t ptr = (size_t) (addr_of_ptr);
   size_t primary_index =  ptr >> 25;
   size_t secondary_index = ((ptr >> 3) & 0x3fffff));

   // trie_root is the base of the first level trie in a global
   secondary_table = trie_root[primary_index];

   // calculate the address of the entry in the secondary trie
   meta_entry = &secondary_table[secondary_index];

  // load base
   ... = meta_entry->base;
  // load bound
   ... = meta_entry->bound;
  // load key
   ... = meta_entry->key;
  // load lock
  ... = meta_entry->lock;
}
```

C code for Metadata Store

```
void sbcets_metadata_store (void* addr_of_ptr,
                            void* base,
                            void* bound,
                            size_t key,
                            void* lock) {

 struct metadata_entry_t* meta_entry;
 struct metadata_entry_t* secondary_table;
 size_t ptr = (size_t) (addr_of_ptr);
 size_t primary_index =  ptr >> 25;
 size_t secondary_index = ((ptr >> 3) & 0x3fffff));

 // trie_root is the base of the first level trie in a global
 secondary_table = trie_root[primary_index];

 // allocate first level entry if NULL
 if(secondary_table == NULL){
    secondary_table = trie_allocate();
    trie_root[primary_index] = secondary_table;
 }

 // calculate the address of the entry in the secondary trie
 meta_entry = &secondary_table[secondary_index];

// store base
 meta_entry->base = base;
// store bound
 meta_entry->bound = bound;
// store key
 meta_entry->key = key;
// store lock
 meta_entry->lock = lock;
}
```

Figure 4.8: C source code for accessing the metadata from the disjoint metadata space with trie lookups. There are 2^{22} entries in the first level trie and there are 2^{23} entries in the second level trie. Metadata is maintained in the second level trie for every 8-byte word.

operations to perform the translation from the program address to the metadata address. Four loads to load the metadata from the second level trie. The metadata store is slightly expensive as it also initializes the first level trie entry if the entry is *NULL* as shown in Figure 4.8. In the common case, the metadata store is fourteen x86 instructions with an extra compare and branch to check if the first level trie entry is *NULL*. By preallocating the secondary level trie entries on memory allocation, these compare and branches can be avoided during metadata store. The x86 code sequences for the metadata load and store are provided in Figure 6.2 of Chapter 6.

Since the second level entries of the trie are allocated only on demand, the memory overhead is significantly lower than either the hashtable and the shadow space. The trie implementation enabled us to experiment with memory-intensive real world programs beyond the few programs described in our SoftBound paper [92]. Metadata lookups to the disjoint metadata space are still expensive. Hence our instrumentation leverages the type information in the intermediate representation to perform such lookups only when pointers are loaded or stored as described in Section 4.2.7.

4.2.3 Instrumentation Handlers for Compiler Instrumentation

To perform pointer-based checking within the compiler, *SoftBoundCETS* instrumentation needs to add extra code into the IR to perform checks, metadata lookups, shadow stack lookups, and metadata propagation. There are two alternatives in introducing such checks. First alternative is to create such handlers in the IR and use instrumentation to inline these handlers at every place similar to inlined reference monitors used for software fault isolation [132]. We take the second alternative where we write the checks, metadata lookups, and shadow stack handlers as C code, which is separately compiled to generate the IR. *SoftBoundCETS* instrumentation just adds calls to the handlers at the appropriate places. Instrumented code is later linked with the compiled IR. We leverage the LLVM compiler's inlining mechanisms to inline the checks and other handlers. This approach has two advantages. First, this approach keeps the instrumentation simple and easy to reason about, which also enabled us to create a verified instrumentation extracted from the proof of correctness in a proof assistant in a related but separate project [137]. Second, this approach separates the implementation choices in metadata organization, memory mapping schemes for the various shadow spaces, and shadow stack operations from the instrumentation over the IR. Our experience suggests such an instrumentation with a division of labor between the actual instrumentation and C handlers could be handy in building a wide range of debugging tools.

4.2.4 The *SoftBoundCETS* Instrumentation Algorithm

Having seen the design choices for the compiler instrumentation in the previous sections and the pointer-based checking approach in Chapter 3, we describe the details of the *SoftBoundCETS* instrumentation in this section. *SoftBoundCETS* instrumentation algorithm for providing pointer-based memory safety operates in three passes. In the first and the second pass, the *SoftBoundCETS* instrumentation gathers the metadata for each pointer value in the program. The third pass introduces spatial safety and temporal safety checks using the metadata gathered in the first two passes. These three passes just make linear passes over the program without the need for any fixpoint algorithms.

Entry block after the first pass of SoftBoundCETS instrumentation

```
define i32 @main(i32 %argc, i8** %argv) nounwind uwtable {
entry:
  call void shadow_stack_store_base(i64 1, i8* NULL);
  call void shadow_stack_store_bound(i64 1, i8* NULL);
  call void shadow_stack_store_key(i64 1, i64 0);
  call void shadow_stack_store_lock(i64 1, i8* NULL);
  %call = call noalias i8* @malloc(i64 16) nounwind
  %call_base =  call i8* shadow_stack_load_base(i64 0);
  %call_bound = call i8* shadow_stack_load_bound(i64 0);
  %call_key = call i64 shadow_stack_load_key(i64 0);
  %call_lock = call i8* shadow_stack_load_lock(i64 0);
                                   Dictionary status
                   %call: <%call_base, %call_bound, %call_key, %call_lock>
  %0 = bitcast i8* %call to %struct.node_t*
                   %call: <%call_base, %call_bound, %call_key, %call_lock>
                   %0 : <%call_base, %call_bound, %call_key, %call_lock>
  %value = getelementptr inbounds %struct.node_t* %0, i32 0, i32 0
                   %call: <%call_base, %call_bound, %call_key, %call_lock>
                   %0: <%call_base, %call_bound, %call_key, %call_lock>
                   %value: <%call_base, %call_bound, %call_key, %call_lock>
  store i64 0, i64* %value, align 8
  %next = getelementptr inbounds %struct.node_t* %0, i32 0, i32 1
                   %call: <%call_base, %call_bound, %call_key, %call_lock>
                   %0: <%call_base, %call_bound, %call_key, %call_lock>
                   %value: <%call_base, %call_bound, %call_key, %call_lock>
                   %next: <%call_base, %call_bound, %call_key, %call_lock>
  call void store_metadata(i8** next,  i8* NULL, i8* NULL, i64 0, i8* NULL );
  store %struct.node_t* null, %struct.node_t** %next, align 8
  br label %for.cond
```

Figure 4.9: Illustration of the first pass of the *SoftBoundCETS* instrumentation for the entry block of function in Figure 4.3.

SoftBoundCETS instrumentation maintains base, bound, key, and lock metadata with every LLVM IR value that has a pointer type. These metadata values are created as temporaries for every LLVM IR instruction that produces a pointer value. These metadata values can be created in two ways: (1) instrumentation inserted calls to the handler functions for pointer loads and metadata retrievals from the shadow stack, and (2) instrumentation created temporaries for propagating the metadata for IR instructions that perform stack allocations (*e.g.* **alloca**) and that manipulate pointers (*e.g.* **getelementptr** and **bitcast**).

```
%call: <%call_base, %call_bound, %call_key, %call_lock>
%0: <%call_base, %call_bound, %call_key, %call_lock>
%value: <%call_base, %call_bound, %call_key, %call_lock>
%next: <%call_base, %call_bound, %call_key, %call_lock>
```

Dictionary status

```
for.cond:
  %ptr.0 = phi %struct.node_t* [ %0, %entry ], [ %1, %for.inc ]
  %ptr.0.base = phi i8* [__, %entry ], [__, %for.inc ]
  %ptr.0.bound = phi i8* [__, %entry], [__, %for.inc ]
  %ptr.0.key = phi i64 [__, %entry], [__, %for.inc ]
  %ptr.0.lock = phi i8* [__, %entry], [__, %for.inc ]
```

```
%call: <%call_base, %call_bound, %call_key, %call_lock>
%0: <%call_base, %call_bound, %call_key, %call_lock>
%value: <%call_base, %call_bound, %call_key, %call_lock>
%next: <%call_base, %call_bound, %call_key, %call_lock>
%ptr.0: <%ptr.0.base, %ptr.0.bound, %ptr.0.key, %ptr.0.lock>
```

```
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i64 %i.0, 128
  br i1 %cmp, label %for.body, label %for.end
```

Figure 4.10: The second basic block (for.cond) of the function in Figure 4.3 after the first pass of *SoftBoundCETS* instrumentation illustrating the handling of phi nodes.

The first pass of the instrumentation traverses each instruction in each basic block in each function in the program in the control flow graph order to introduce metadata temporaries. The instrumentation maintains a dictionary that maps a pointer temporary to its corresponding metadata. In the entry block of a function, *SoftBoundCETS* instrumentation introduces shadow stack loads to retrieve the metadata whenever the function has pointer arguments. Similarly when there is an external call, it introduces shadow stack loads/stores to setup the metadata. For other pointer manipulating instructions such as **getelementptr** and **bitcast**, it simply updates the dictionary entry of the temporary being produced to be equal to the dictionary entry of the source operand of these instructions. For pointer loads, the instrumentation introduces calls to the C handlers that load the appropriate metadata in a temporary. For **store** instructions that store a pointer to memory, the instrumentation introduces calls to the C handlers to the store the metadata. Figure 4.9 illustrates the metadata temporaries and calls introduced by the *SoftBoundCETS* instrumentation for the entry

Entry block after complete SoftBoundCETS instrumentation

```
define i32 @main(i32 %argc, i8** %argv) nounwind uwtable {
entry:
    call void shadow_stack_store_base(i64 1, i8* NULL);
    call void shadow_stack_store_bound(i64 1, i8* NULL);
    call void shadow_stack_store_key(i64 1, i64 0);
    call void shadow_stack_store_lock(i64 1, i8* NULL);
  %call = call noalias i8* @malloc(i64 16) nounwind
    %call_base =  call i8* shadow_stack_load_base(i64 0);
    %call_bound = call i8* shadow_stack_load_bound(i64 0);
    %call_key = call i64 shadow_stack_load_key(i64 0);
    %call_lock = call i8* shadow_stack_load_lock(i64 0);
                                        Dictionary status
                      %call: <%call_base, %call_bound, %call_key, %call_lock>
  %0 = bitcast i8* %call to %struct.node_t*
                      %call: <%call_base, %call_bound, %call_key, %call_lock>
                      %0: <%call_base, %call_bound, %call_key, %call_lock>
  %value = getelementptr inbounds %struct.node_t* %0, i32 0, i32 0
                      %call: <%call_base, %call_bound, %call_key, %call_lock>
                      %0: <%call_base, %call_bound, %call_key, %call_lock>
                      %value: <%call_base, %call_bound, %call_key, %call_lock>
    call void spatial_check ( i64* value, i8* %call_base, i8* %call_bound, i64 8);
    call void temporal_check(i64 %call_key, i8* %call_lock);
  store i64 0, i64* %value, align 8
  %next = getelementptr inbounds %struct.node_t* %0, i32 0, i32 1
                      %call: <%call_base, %call_bound, %call_key, %call_lock>
                      %0: <%call_base, %call_bound, %call_key, %call_lock>
                      %value: <%call_base, %call_bound, %call_key, %call_lock>
                      %next: <%call_base, %call_bound, %call_key, %call_lock>
    call void store_metadata(i8** next,  i8* NULL, i8* NULL, i64 0, i8* NULL );
    call void spatial_check (%struct.node_t**next, i8* %call_base, i8* %call_bound, i64 8);
    call void temporal_check(i64 %call_key, i8* %call_lock);
  store %struct.node_t* null, %struct.node_t** %next, align 8
  br label %for.cond
```

Figure 4.11: The entry block of the function in Figure 4.3 after the three passes of the *SoftBound-CETS* instrumentation.

```
%call: <%call_base, %call_bound, %call_key, %call_lock>
%0: <%call_base, %call_bound, %call_key, %call_lock>
%value: <%call_base, %call_bound, %call_key, %call_lock>
%next: <%call_base, %call_bound, %call_key, %call_lock>
%1: <%call1_base, %call1_bound, %call1_key, %call1_lock>

                                          Dictionary status
for.cond:
  %ptr.0 = phi %struct.node_t* [ %0, %entry ], [ %1, %for.inc ]
  %ptr.0.base = phi i8* [%call_base, %entry ], [ %call1_base, %for.inc ]
  %ptr.0.bound = phi i8* [%call_bound, %entry], [%call1_bound, %for.inc ]
  %ptr.0.key = phi i64 [%call_key, %entry], [%call1_key, %for.inc ]
  %ptr.0.lock = phi i8* [%call_lock, %entry], [%call1_lock, %for.inc ]
    %call: <%call_base, %call_bound, %call_key, %call_lock>
    %0: <%call_base, %call_bound, %call_key, %call_lock>
    %value: <%call_base, %call_bound, %call_key, %call_lock>
    %next: <%call_base, %call_bound, %call_key, %call_lock>
    %ptr.0: <%ptr.0.base, %ptr.0.bound, %ptr.0.key, %ptr.0.lock>
    %1: <%call1_base, %call1_bound, %call1_key, %call1_lock>
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i64 %i.0, 128
  br i1 %cmp, label %for.body, label %for.end
```

Figure 4.12: The second basic block (for.cond) of the function in Figure 4.3 after the three passes of *SoftBoundCETS* instrumentation with fully populated phi nodes.

---

block of the function in Figure 4.3 after the first pass of metadata propagation. The state of the dictionary is displayed in the figure after each dictionary update. The instrumentation does nothing for the **store** instruction in line 6 in Figure 4.9 as it does not store any pointer values.

SoftBoundCETS instrumentation performs two passes to gather metadata to handle circular dependences with **phi** nodes. The first pass creates metadata **phi** nodes for each original **phi** node that has a pointer type in the IR. The inputs to the metadata **phi** nodes are not populated in the first pass as the inputs to the pointer **phi** node may be defined later in the traversal order or in the same basic block as the **phi** node. The first pass ends when all the basic blocks are traversed. Figure 4.10 illustrates the metadata **phi** nodes introduced by the instrumentation for the second basic block (for.cond) in the function from Figure 4.3. Figure 4.10 shows that the inputs to the metadata **phi** nodes are empty after the first pass of the instrumentation. In the second pass, the instrumentation traverses the basic blocks and populates the inputs to the **phi** nodes.

In the third pass, the instrumentation introduces a call to the C handlers that perform the spatial and temporal checks using the metadata temporaries. We defer the check introduction to the third pass over the program as it enables us to decouple the metadata propagation and the check optimizations to eliminate redundant checks. Figure 4.11 and Figure 4.12 show the instrumented IR code after all three passes of the *SoftBoundCETS* instrumentation for the entry block and the for.cond block of the function in Figure 4.3.

4.2.5 Advantages of Compiler Instrumentation

We discuss the advantages of performing *SoftBoundCETS* instrumentation on the LLVM IR described above in the previous sections. There are two primary advantages provided by structuring our instrumentation over the LLVM IR. First, as all optimizations preserve the semantics of the LLVM IR, *SoftBoundCETS* can instrument optimized code minimizing the amount of extra instrumentation code added. Second, explicit types in the SSA-based IR enables us to identify pointers. Further, we can leverage the compiler's built-in analyses to perform various optimizations. We discuss these issues in the following sections.

4.2.6 Instrumenting Optimized Code

The first primary advantage of performing the *SoftBoundCETS* instrumentation within the compiler is the ability to leverage other compiler optimizations and analyses. The compiler optimizations in the LLVM compiler preserve the type information and the semantics of the original program. Hence, the compiler-based memory safety instrumentation can be performed on optimized code. Further as register promotion and other common optimizations have already reduced the number of memory operations, the strategy of instrumenting optimized code reduces the amount of additional instrumentation introduced. This ability to instrument optimized code and still provide comprehensive detection of memory safety errors subsumes the benefits of a source-code based instrumentation. Further every compiler has a large suite of analyses which can be used to further optimize the memory safety transformation. We describe few of these optimizations to streamline *SoftBoundCETS* instrumentation in Section 4.3.

4.2.7 Pointer Identification

One of the primary reasons that we were able to implement pointer-based checking within the compiler as described in the previous sections is attributed to the type information in the LLVM IR. The

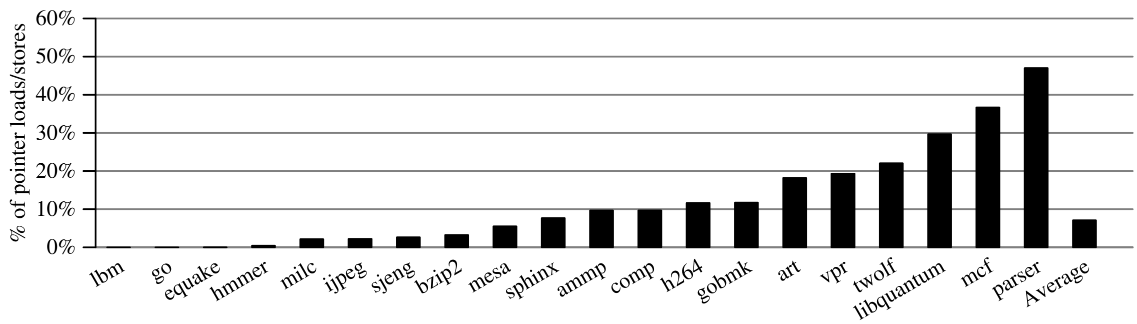


Figure 4.13: The percentage of memory operations that load or store a pointer from/to memory, thus requiring a metadata access.

---

type information in the LLVM IR is used to identify pointers. Using the type information in the LLVM intermediate representation (IR), we overlay a system with two types — pointer types and non-pointer types — on top of the LLVM IR types. We perform the metadata creation and propagation operations only for the operations with pointer types. Among the instructions in the LLVM IR instruction set presented in Figure 4.1, memory accessors (**load**, **store**), stack allocation (**alloca**), pointer conversion operators (**ptrtoint**, **inttoptr**, **bitcast**), **getelementptr**, **phinode**, **call**, and **ret** instructions produce or manipulate pointer values. Our instrumentation maintains metadata only for instructions producing or manipulating pointer values.

As the LLVM IR is in SSA form, the majority of the instructions produce or manipulate pointer values in temporaries without accessing memory. Hence, our memory safety instrumentation maintains the metadata for such IR instructions in temporaries. We leverage the LLVM's machine dependent register allocation to eventually map these temporaries to machine register or introduce spill code. All memory accesses in LLVM IR occurs through explicit load and store operations. All such memory accesses are conceptually checked for spatial and temporal safety violations. Only when load (store) IR instructions load (store) a pointer value, the disjoint metadata discussed in Section 3.8 of Chapter 3 is accessed by introducing calls to the C handlers as described above.

First, a compiler instrumentation that performs instrumentation only for IR instructions with pointer types provides memory safety even in the presence of arbitrary type casts in the program among pointer types and casts from pointer types to non-pointer types. Second, it reduces the amount of instrumentation performed as extra instrumentation is added only for pointer operations.

In the absence of pointer information, every memory access would load/store the metadata, which would have introduced significant overhead.

To measure the benefits of using the pointer information in the LLVM IR to identify pointers loads/stores, we instrumented the LLVM IR to count the number of loads/stores and pointer loads/-stores in the dynamic execution. Figure 4.13 reports the percentage of dynamic memory operations that load or store a pointer value. The benchmarks in the Figure 4.13 are sorted in the increasing order of pointer load/store percentages. The benchmarks on the right of the figure have a larger percentage of pointer loads/stores requiring accesses to the disjoint metadata space. An important point to note that these dynamic memory operations do not count the pointers that could be loaded/-stored as part of spill code, which is absent when the *SoftBoundCETS* performs its instrumentation. The majority of the SPEC benchmarks have metadata access ratios of less than 10%. On average, only 7% of the memory accesses load/store a pointer, highlighting the benefits of leveraging the type information in the IR.

Limitation of *SoftBoundCETS* A limitation of just propagating metadata with the pointers is that we disallow the creation of pointers from non-pointers (*e.g.* creating a pointer from an integer). By default, such manufactured pointers get null bounds with an invalid identifier as the metadata and the program execution is stopped when such pointers are dereferenced. Although we have not seen such code in our experimentation, the programmer is required to provide valid metadata if such programs are required to execute without memory safety violations.

4.2.8 Other Instrumentation Considerations

Beyond the issues described in implementing pointer-based checking in Chapter 3 and prior sections in this chapter, there are many issues that we address to handle real world applications. We describe these issues below.

Handling Libraries through Wrappers *SoftBoundCETS* instrumentation can be used with existing library source code to generate memory-safe libraries. Such libraries can be later linked with other code instrumented with *SoftBoundCETS* to create memory-safe binaries. In cases where the library code is not available, libraries can interface with *SoftBoundCETS* instrumented code using wrappers. Figure 4.14 illustrates the wrappers for two functions of the string library: *strchr* and

Prototype for strchr

```
char * strchr(const char *s, int C);
```

SoftBoundCETS wrappers for strchr

```
char * sbcets_strchr(const char *s, int C){
  char* ret_ptr = strchr(s, c);
  if(ret_ptr == NULL){
   __sbcets_store_invalid_return_metadata();
  }
  else{
   __sbcets_propagate_metadata_shadow_stack_from(1, 0);
  }
  return ret_ptr;
}
```

Prototype for strtol

```
long strtol(const char *nptr, char** endptr, int base);
```

SoftBoundCETS wrappers for strchr

```
long sbcets_strtol(const char *nptr, char** endptr, int base){
  long tmp = strtol(nptr, endptr, base);
  if(endptr != NULL){
    __sbcets_propagate_shadow_stack_to_address(endptr, 1);
  }
  return tmp;
}
```

Figure 4.14: Wrappers for *SoftBoundCETS* instrumentation.

strtol. The library function *strchr* searches for the occurrence of the second argument (*i.e.*, c) in the input string, which is the first argument. If the string contains the character then it returns the pointer to the first occurrence of the character and *NULL* otherwise. To interface with *SoftBoundCETS*, the library code should return metadata for the pointer being returned, otherwise the program will experience false violations. The wrappers provide such glue code to interface with un-instrumented library code. The wrapper for function *strchr* updates the return metadata based on the actual value being returned. If the return value is NULL, then it stores invalid metadata for the returned pointer in the shadow stack. Otherwise, the wrapper just propagates the metadata of the first argument in the shadow stack to the returned pointer. *SoftBoundCETS* instrumentation provides handlers (*e.g.*, *__sbcets_store_invalid_return_metadata*) as illustrated in Figure 4.14 to make such interfacing easier. Further, these wrappers also perform spatial and temporal safety checks according to the library specification to detect memory safety errors triggered in the library code (such checks have been omitted from the Figure 4.14 for simplicity).

Libraries can also return pointers to the instrumented code by storing the pointers in memory locations. Such memory locations can be subsequently accessed to retrieve the pointer. The wrapper for the library function *strtol* is provided in Figure 4.14 to demonstrate one such example of pointers being returned from libraries using memory locations. The library function *strtol* converts the initial part of the string in the first argument *nptr* to a long integer value according to the given base. When the input string has an invalid digit with respect to a particular base, the library stores the pointer to the invalid digit in the memory location pointed by the second argument (*i.e.*, *endptr*) to *strtol*. The wrapper updates the disjoint metadata with the metadata of the input argument as illustrated in the wrapper in the Figure 4.14 using the *SoftBoundCETS* handler (*__sbcets_propagate_shadow_stack_to_address*). Although, writing such wrappers can be tedious, our instrumentation library provides various handlers and wrappers to make this job easier. Further, disjoint metadata alleviates this problem of writing wrappers by eliminating the need for deep copies of data structures.

Memcpy() Among various C standard library calls, `memcpy` requires special attention. First, to reduce runtime overhead, the source and targets of the `memcpy` are checked for spatial safety and temporal safety once at the start of the copy. Second, `memcpy` must also copy the metadata corresponding to any pointer in the region being copied. By default, *SoftBoundCETS* takes the safe but slow approach that inserts code to copy the metadata for all memcopies. However, most calls to *memcpy* involve buffers of non-pointer values. To address this inefficiency, *SoftBoundCETS* infers whether the source and destination of the `memcpy` do not contain pointers by looking at the type of the argument at the call site. We have found this approach facilitated by the type information in the LLVM IR sufficient to identify the few uses of `memcpy` involving pointers in our benchmarks.

Global Initialization For global allocations such as global arrays, the base, bound, key and lock metadata are compile-time constants. Thus, *SoftBoundCETS* sets these bounds without requiring writing the metadata to memory. However, for pointer values that are in the global space and are initialized to non-zero values, *SoftBoundCETS* adds code to explicitly initialize the in-memory memory metadata for these variables. This is implemented using the same hooks C++ uses to run code for constructing global objects (*i.e.* by appending *__attribute__((constructor))* to the definition of the function that initializes the metadata).

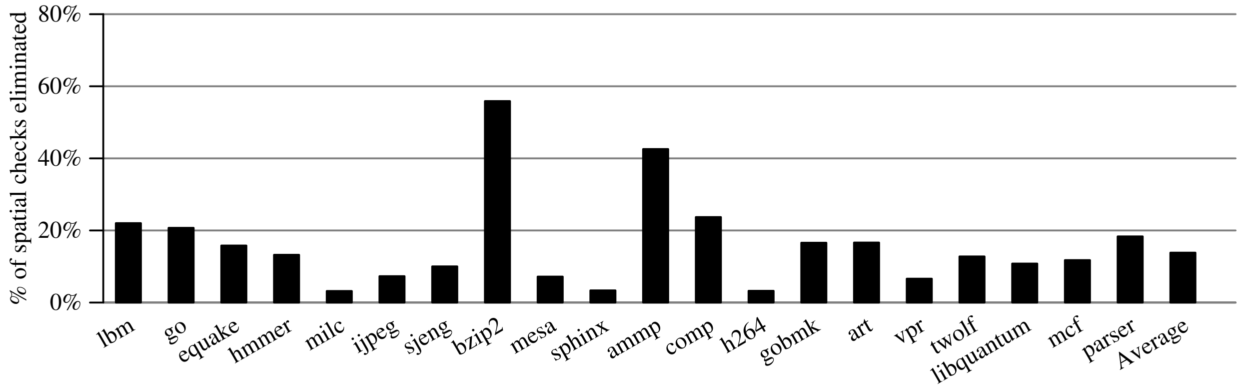


Figure 4.15: Percentage of redundant **spatial** checks removed.

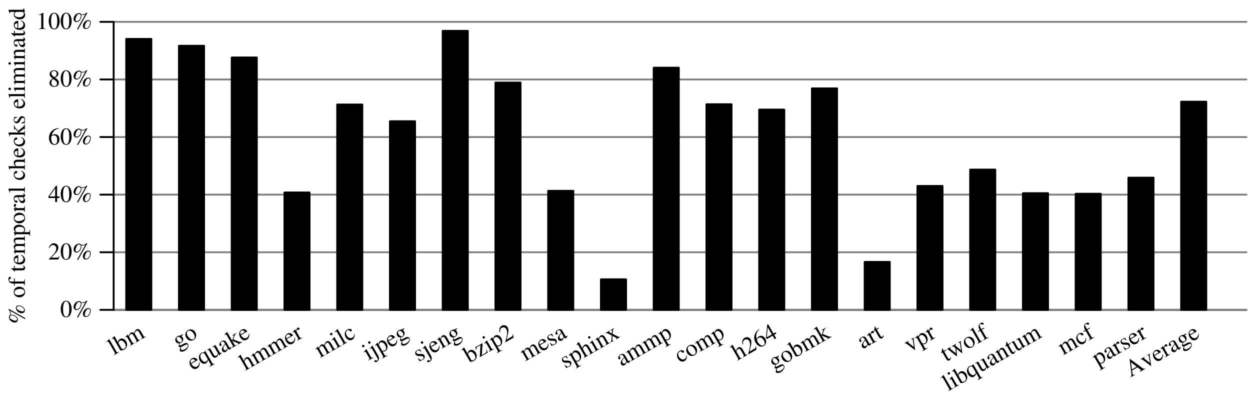


Figure 4.16: Percentage of redundant **temporal** checks removed.

4.3 Optimizations to *SoftBoundCETS*

The *SoftBoundCETS* instrumentation described above in this chapter can be further optimized to reduce the performance overheads by (1) eliminating redundant spatial and temporal checks and (2) eliminating unnecessary checks when the program is type-safe.

4.3.1 Implementing Custom Check Elimination

Bounds check elimination is a well-established and long-studied problem [23, 62, 130]. *SoftBound-CETS* instrumentation can leverage the existing analyses such as dominator tree construction, alias analysis, and loop invariant code motion in the compiler to perform these optimizations. Our check elimination optimizations broadly falls under two categories: removal of unnecessary checks and removal of redundant checks.

Unnecessary Check Elimination *SoftBoundCETS* instrumentation occurs on an intermediate representation (IR) with infinite registers, spills and restores are not visible at the IR level, so spatial and temporal checks are simply not inserted. Similarly spatial and temporal checks are not necessary for accesses to global scalar values and constants. Further, no temporal checks are required for any pointer that is directly derived from the stack pointer within the corresponding function call, because the stack frame is guaranteed to live until the function exits. In the same vein, checking stack spills and restores is unnecessary. In addition, performing temporal checks to pointers known to point to global objects is unnecessary, because global objects are never deallocated. Our instrumentation uses a simple dataflow analysis to identify these pointers and elides their checks.

Redundant Check Elimination A spatial check is redundant if there is an earlier check to the same pointer that dominates the current memory access. We use the existing dominator tree provided by the compiler infrastructure to identify such spatial checks and elide them. A temporal check is redundant if: (1) there is an earlier check to pointers that share the same values for key and lock address (*i.e.*, points to the same object) and (2) the check is not killed by a call to *free*. Our algorithm for finding redundant temporal checks constructs a dominator tree and uses a standard dataflow analysis to identify the root object accessed by each reference. Any check dominated by an earlier check that is not killed along the path is removed. In our current implementation a temporal check is killed when there is any function call along the path—that is, we conservatively assume that any function could potentially *free* any pointer.

Check Removal Effectiveness The graphs in Figure 4.15 and Figure 4.16 show the percentage of runtime spatial and temporal checks eliminated as a percentage of runtime checks performed by *SoftBoundCETS* without such redundant-check optimizations—taller bars are better. On average, 14% of the spatial checks have been eliminated by redundant check optimizations. On average, 72% of the temporal checks have been eliminated by redundant check optimizations.

Figure 4.16 demonstrates that our check optimizations are more effective in removing the temporal checks than the spatial checks. We illustrate the reason for this effectiveness in temporal check

```
if ( ptr1_key != *( ptr1_lock ))  { abort();  }
... = *(ptr1 + offset1);
...
...
*(ptr2) = ...; // potentially aliasing store
...
if ( ptr1_key != *( ptr1_lock ))  { abort(); }
... = *(ptr1 + offset2);
```

Figure 4.17: Optimizing redundant temporal checks.

elimination with an example. Consider the code snippet in Figure 4.17. Naive temporal check insertion would insert two temporal checks (shaded). Our temporal-check elimination optimization removes the second check as redundant, because it knows that (1) the checks are to the same object and (2) the intervening code does not kill the validity of temporal check (a store can not deallocate memory and thus cannot cause *ptr1* to dangle). The store to *ptr2* however blocks standard compiler optimizations from removing the redundant loads of **ptr1_lock*. However, if the store was not present (or with better alias analysis) standard compiler optimizations alone would easily eliminate the second temporal check.

This example also illustrates two key differences between spatial and temporal check elimination. First, even though the two loads are to different addresses, the temporal check is redundant whereas a bounds check would generally not be redundant. Second, a function call between the two loads would block a temporal check removal, whereas the redundancy of spatial check to the same address is independent of the intervening code.

4.3.2 Optimizing Type-Safe Programs

SoftBoundCETS instrumentation conceptually checks every memory access to provide spatial safety and temporal safety. Such checking is required to provide memory safety in the presence of arbitrary type casts as described in Chapter 3. In this section, we explore opportunities for optimizing the instrumentation when the programs are type-safe with respect to pointer operations. The program can still use type-casts with non-pointer values. These optimizations have been influenced by prior research on using type information to improve compilation [90] and provide memory safety [46, 94].

When a program is type-safe, we optimize the instrumentation to eliminate spatial safety checks on accesses to sub-fields in a structure. An important thing to note is that these optimizations for type-safe programs only reduce the overhead of enforcing spatial safety but not temporal safety. The C language allows arbitrary type-casts. So we perform the optimizations described in this chapter only when our analysis can determine that there are no casts involving pointers in the program.

Figure 4.18 briefly presents the algorithm for performing spatial check optimizations for type safe programs. To identify that the program is type-safe, the algorithm iterates over all the instructions in the module looking for bitcast instructions. All type casts occur in the LLVM IR using bitcast instructions. Algorithm inspects the source and destination types of the *bitcast* instruction to check whether they are pointers. If there exists a pointer cast, then the program is type unsafe with respect to pointer operations and no optimizations are performed.

If the program does not have any bitcast operations in the LLVM IR involving pointer types, then it optimizes the spatial checks for the structure sub-fields. The algorithm iterates over every load instruction and store instruction in the module. For each such load and store instruction, it does a backward analysis to check if the operand is a GetElementPtrInst with constant indices and with input operand having a struct type. In such cases, it elides the spatial checks for the load and store instructions. The rationale in requiring the GetElementPtr indices to be constant is to ensure that array access with a non-constant index are checked.

These optimizations for type-safe programs are optional and are turned off by default. Programmer can invoke these optimizations by explicitly providing a command line flag. When the program is type-unsafe, the compilation will perform the default *SoftBoundCETS* instrumentation.

4.4 Performance Evaluation

In this section, we describe and experimentally evaluate our prototype implementation of *SoftBoundCETS*. The goal of this evaluation is to measure its runtime overhead, sources of inefficiencies, and identify the inefficiencies that can be isolated to be accelerated with hardware support.

4.4.1 Prototype

LLVM is a moving target. We have made the *SoftBoundCETS* instrumentation available for LLVM versions from version 2.4 to version 3.0. We use LLVM version 3.0 for all the results described

Check Program Pointer Type Safe

Step #1. Identify the bitcast operations in the LLVM IR and check if there any casts involving pointer types to structures.

```
foreach Function F in Module M:
  foreach BasicBlock B in Function F:
    foreach Instruction I in BasicBlock B:
      if I is a bitcast instruction
        if source and destination types are pointers to struct types
          return TypeUnsafe
```

Step #2. For every load and store operation, identify if it is derived from a pointer to a structure type and elide checks:

```
foreach LoadInst Ld and StoreInst St in Module M:
  if pointer_operand(Ld or St) is a GetElementPtrInst
    if GetElementPtr's pointer operand is a struct type
      if GetElementPtr indices are all constants
        Eliminate spatial safety checks for Ld and St
```

Figure 4.18: **Optimizations for Type Safe Programs.** These optimizations occur by searching the module for bitcast operations between pointer operations. Then when the program is type safe, it optimizes the spatial checks to structure field accesses.

in this dissertation. The *SoftBoundCETS* pass operates on LLVM's ISA-independent intermediate form, so the *SoftBoundCETS* pass is independent of any specific ISA. We selected 64-bit x86 as the ISA for evaluation due to its ubiquity. The *SoftBoundCETS* pass is approximately 6K lines of C++ code. The *SoftBoundCETS* instrumentation source code is publicly available [6]. The *SoftBound-CETS* instrumentation is also integrated with the Clang compiler driver in LLVM's experimental subversion trunk maintained by the SAFECode project.

4.4.2 Benchmarks

We used C benchmarks selected from the SPECINT and SPECFP benchmark suites (from 1995, 2000 and 2006 versions) to evaluate *SoftBoundCETS*'s performance. We use an an enhanced version of the *equake* benchmark that uses a proper multidimensional array and thus improves its baseline performance by 60%. We ran all the SPEC benchmarks with the reference inputs provided by the suite. All runs are performed on a 3.4 Ghz Intel Core i7 (SandyBridge) processor. We used Intel's Pin tool to count the dynamic instructions and instruction mix. We ran each benchmark multiple times to eliminate experimental noise.

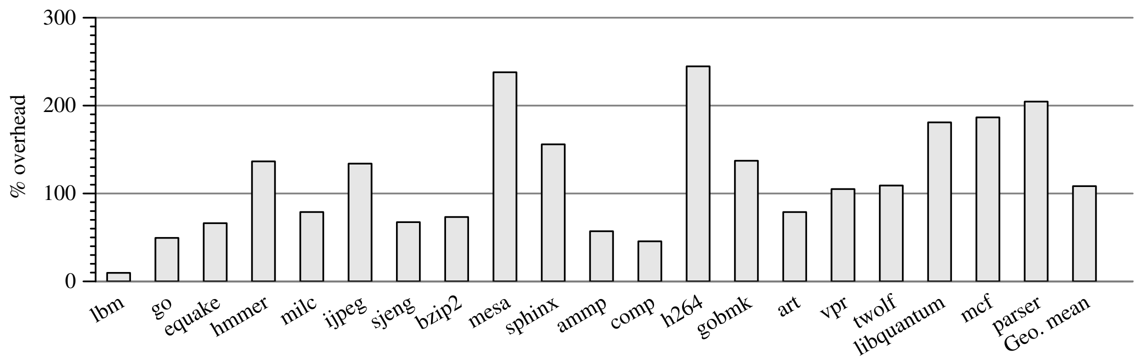


Figure 4.19: Execution time performance overhead of *SoftBoundCETS* with full checking over a fully optimized uninstrumented baseline.

4.4.3 Runtime Overheads of Enforcing Spatial and Temporal Safety

Figure 4.19 presents the percentage runtime overhead of *SoftBoundCETS* instrumentation over an uninstrumented baseline (smaller bars are better as they represent lower runtime overheads). The total height of each bar corresponds to the overhead of full checking *i.e.*, enforcing both spatial and temporal safety with all the optimizations except the type-safe program optimizations. The average runtime overhead is 108%. The performance overheads are highly correlated with the percentage of dynamic pointer loads and stores in the program shown in 4.13. The benchmarks on the left in Figure 4.19 that have lower frequency of metadata accesses generally have lower runtime overheads. On those benchmarks the overhead is largely due to bounds and lock-key checking. In contrast, the benchmarks on the right have a larger frequency of pointer loads/stores that access the metadata space. Hence, the benchmarks on the right incur overheads to perform metadata accesses along with checking overhead.

Store-only Checking and Propagation-only Overheads Figure 4.20 presents the execution time overhead of performing *SoftBoundCETS* with (1) full-checking (leftmost bar), (2) store-only checking (middle bar) and (3) metadata propagation both in registers and memory but without any checking (rightmost bar). Checking only stores can prevent security vulnerabilities as described in Chapter 3. The height of the middle bar in Figure 4.20 presents the overheads for checking the bounds and identifiers only for store operations and propagating the metadata on all other operations. Checking only stores reduces the number of check operations. However, the number of metadata writes is

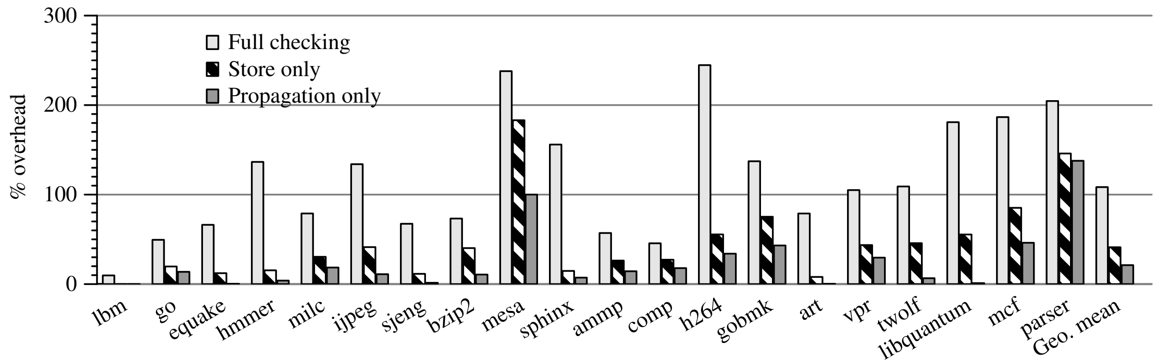


Figure 4.20: Execution time performance overhead of *SoftBoundCETS* with full checking, store-only checking, and propagation-only over a fully optimized uninstrumented baseline.

---

unchanged, because all the same pointer metadata must be propagated through memory to check subsequent store dereferences. When using aggressive LLVM's dead code elimination, many of the metadata reads are also removed (those that feed only load dereference checks). The runtime overhead of store-only checking is 41% on average for our benchmarks.

The benchmarks on the left that have very few pointer loads/stores attain significant performance benefit with store-only checking. For benchmarks on the left of Figure 4.20 eliminating load dereference checks (both spatial and temporal) reduces the primary source of overhead, *i.e.* checks. For the benchmarks on the right, that have frequent metadata accesses, store-only checking does provide reasonable benefit by eliminating the load dereference checks and relying on the LLVM optimizer to remove the metadata loads that just feed the load dereference checks.

The height of the rightmost bar of each benchmark in Figure 4.20 presents the overhead of performing metadata propagation without any checking. LLVM's aggressive optimizer removes metadata loads that just feed the checks through dead-code elimination. However, metadata loads that feed other metadata stores or shadow stack operations are not optimized. The height of the bar also includes the cost of metadata propagation using the shadow stack. On average, propagating the metadata and other auxiliary operations introduced by the *SoftBoundCETS* without any checks incurs a performance overhead of 21%. The benchmarks on the right of the Figure 4.20 that perform frequent metadata accesses report higher overheads even with just metadata propagation. Although benchmark *mesa* has reasonably lower number of metadata loads/stores (on the left in Figure 4.20),

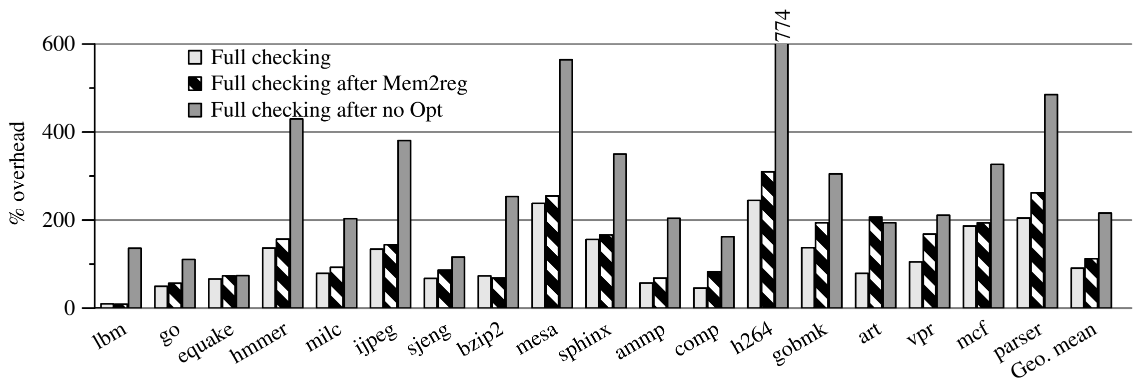


Figure 4.21: Execution time performance overheads of *SoftBoundCETS* instrumentation when the instrumentation is performed without any compiler optimizations (Full checking after no opt) and with *Mem2reg* optimization (Full checking after *Mem2reg*). The configurations are fully optimized again after the *SoftBoundCETS* instrumentation.

---

it suffers high overhead even with metadata propagation only configuration as a result of frequent shadow stack operations.

4.4.4 Benefits of Compiler Instrumentation

Figure 4.21 presents the benefits of performing *SoftBoundCETS* instrumentation on optimized code leveraging the existing optimizations within the compiler. There are the three bars for each benchmark. The height of the leftmost bar is the *SoftBoundCETS* instrumentation performed on optimized code, which is same as the bar in Figure 4.19. The height of the right-most bar represents the overhead of *SoftBoundCETS* when the instrumentation is carried out on unoptimized code. The resultant code after the *SoftBoundCETS* instrumentation is optimized again. This configuration approximates the overheads of memory safety transformation performed on C source code instead on the LLVM IR. The average overhead of *SoftBoundCETS* on unoptimized IR code is 220% on an average, which is more than double the average overhead of *SoftBoundCETS* on optimized code (108%).

The overheads of *SoftBoundCETS* on unoptimized code is a lot higher because LLVM takes unorthodox approach to producing IR code. LLVM generates IR code that is trivially in SSA form by placing all local variables and temporaries in stack locations. As a result, there is a large increase in the number of memory operations. All these memory operations are conceptually checked by *SoftBoundCETS* instrumentation. LLVM provides the *Mem2reg* optimization pass that is responsi-

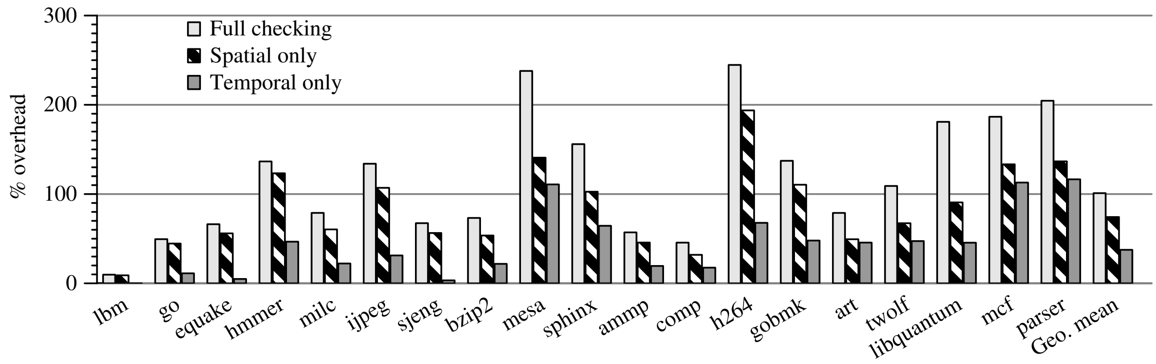


Figure 4.22: Execution time performance overhead for providing spatial safety and temporal safety separately in contrast to full memory safety.

---

ble for promoting un-aliased local variables and stack-based temporary values into registers. The *Mem2reg* optimization uses the standard SSA construction algorithm by Cytron *et al.* [41].

To evaluate the reduction in performance overhead due to the *Mem2reg* optimization (SSA construction), the height of the middle bar in each benchmark in Figure 4.21 reports the overhead of *SoftBoundCETS* instrumentation performed on code optimized with the *Mem2reg* optimization. The overhead of *SoftBoundCETS* instrumentation on code optimized with the *Mem2reg* optimization is considerably lower compared to *SoftBoundCETS* instrumentation on unoptimized code. On average, the overhead of *SoftBoundCETS* instrumentation is reduced from 220% (with instrumentation on unoptimized code) to 120%. The difference between the height of the leftmost and the middle bars in Figure 4.21 presents the benefit of performing *SoftBoundCETS* instrumentation on fully optimized code. Benchmarks such as *art*, *h264* and *vpr* benefit significantly by running LICM (loop invariant code motion) and inlining optimizations before the *SoftBoundCETS* instrumentation.

4.4.5 Enforcing Spatial-only Safety and Temporal-only Safety

Figure 4.22 reports the execution runtime overhead of providing spatial safety alone and temporal safety alone in contrast to full memory safety. There are three bars for each benchmark. The height of the middle bar for each benchmark represents the overhead of providing just spatial safety. There are only 16 bytes of metadata associated with each pointer in the disjoint metadata space for this configuration. The average runtime overhead of enforcing just spatial safety is 74% for our

Benchmark	Valgrind	SAFECode	*SoftBoundCETS*
lbm	26.52×	4.67×	1.09×
go	21.12×	28.41×	1.49×
equake	30.15×	22.93×	1.66×
hmmer	26.72×	71.55×	2.36×
milc	18.93×	21.52×	1.78×
ijpeg	27.84×	48.23×	2.33×
sjeng	35.08×	25.30×	1.67×
bzip2	30.53×	32.72×	1.73×
mesa	47.34×	70.44×	3.37×
sphinx	37.00×	34.04×	2.55×
ammp	26.61×	23.89×	1.57×
h264	38.36×	59.45×	3.44×
art	27.31×	39.24×	1.78×
vpr	43.94×	31.45×	2.11×
twolf	26.38×	34.04×	2.11×
libquantum	15.71×	24.88×	2.80×
mcf	6.77×	12.63×	2.86×
parser	36.54×	24.10×	3.04×
Avg	**26.53**×	**20.83**×	**2.08**×

Table 4.1: Execution time overheads other publicly available tools such as Valgrind's Memcheck [116] and SAFECode [46] in contrast to *SoftBoundCETS*. We have omitted benchmarks that we were not able to run with all the three tools.

---

benchmarks. Similarly the height of the rightmost bar of each benchmark in Figure 4.22 represents the overhead of just providing temporal safety. The average runtime overhead of enforcing just temporal safety is 40% on average for these benchmarks. These overheads are lower than the overhead reported in prior work on CETS [93]. The differences in overheads are attributed to the new optimizations (described in Section 4.3), streamlined design and a more robust prototype compared to the earlier versions of the prototype.

4.4.6 Comparison With Other Approaches

Table 4.1 presents the execution time performance overhead of running other publicly available tools with our benchmarks. We use two publicly available tools to perform this comparison: (1) Valgrind-3.7's Memcheck tool that implements a tripwire approach described in Chapter 2 to detect memory errors on the heap, and (2) SAFECode project for LLVM-3.0 from the experimental LLVM

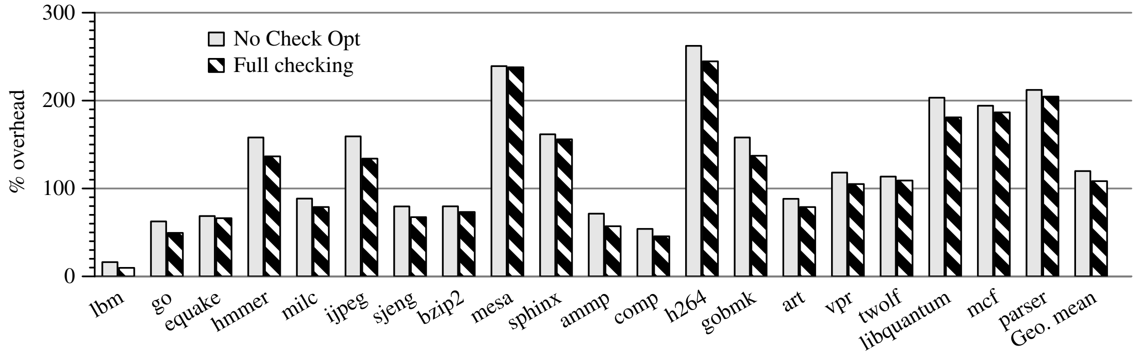


Figure 4.23: Execution time performance overheads of *SoftBoundCETS* instrumentation when the the checks are optimized with the custom check elimination described in Section 4.3.1.

---

trunk that performs object-based checking described in Chapter 2. Among the two tools, SAFECode tool performs more precise spatial checking similar to *SoftBoundCETS* when compared to the Valgrind's Memcheck tool. Similar to *SoftBoundCETS*, SAFECode operates on the LLVM IR. We observe that both Valgrind's Memcheck tool and SAFECode slow down the program by an order of magnitude more than *SoftBoundCETS*. Memcheck tool's average slowdown is 26.53× and SAFECode's average slowdown is 20.83× when compared to the baseline[1]. In contrast, *SoftBoundCETS* incurs an average slowdown of just 2.08× (108% performance overhead).

4.4.7 Impact of Custom Check Elimination

Figure 4.23 presents the execution time overhead of *SoftBoundCETS* without and with custom check elimination described in Section 4.3.1. The height of the left bar represents the performance overhead of *SoftBoundCETS* without custom check elimination. The height of the right bar is the performance overhead of *SoftBoundCETS* with custom check elimination. On average, custom check elimination reduces the the average from 120% to 108%. The reduction in performance overhead with custom check elimination is moderate in spite of removing a significant fraction of the checks as described in Section 4.3.1. This behavior is a result of using an aggressive LLVM optimizer to optimize the resultant code after *SoftBoundCETS* instrumentation, which probably removes many of the checks removed by the custom check elimination.

[1]Publications from the SAFECode project used Automatic Pool Allocation and report substantially lower overheads than the 20.83x overhead. However, automatic pool allocation is currently not integrated with SAFECode in the release for LLVM-3.0.

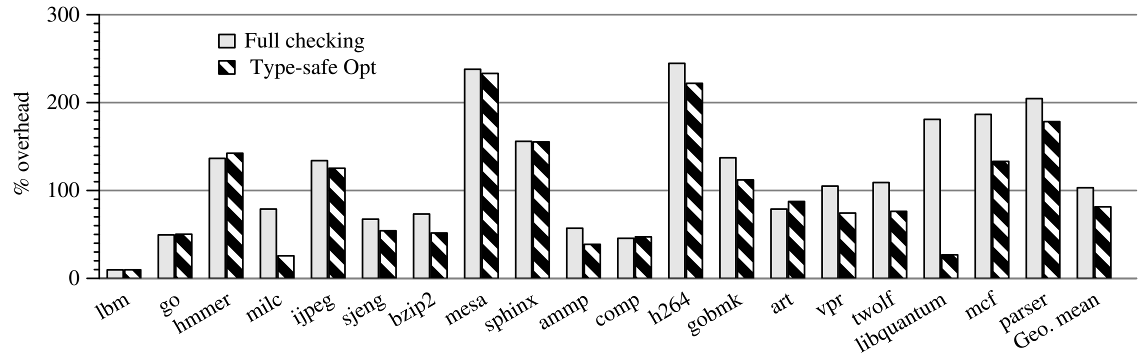


Figure 4.24: Execution time performance overheads of *SoftBoundCETS* instrumentation when the spatial checks are eliminated for type-safe programs as described in Section 4.3.2.

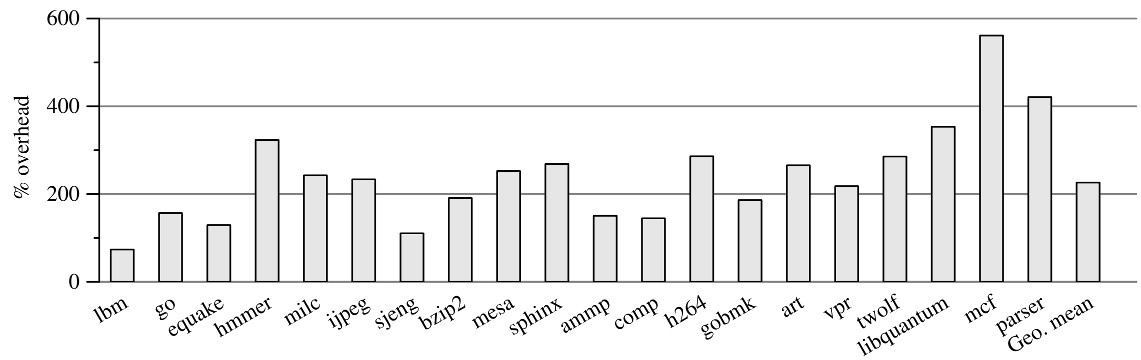


Figure 4.25: Instruction overheads with *SoftBoundCETS*.

4.4.8 Impact of Custom Check Elimination with Type-Safe Programs

Figure 4.24 presents the execution time overhead of *SoftBoundCETS* without and with spatial check elimination for type-safe programs described in Section 4.3.2. The height of the left and the right bar represent the performance overhead of *SoftBoundCETS* without (default mode) and with spatial check elimination for type-safe programs respectively. Eliminating spatial checks for type-safe programs can reduce the overhead significantly for some benchmarks (*e.g.*, *libquantum*). On average, eliminating spatial checks for type-safe programs reduces the performance overhead from 108% to 81%.

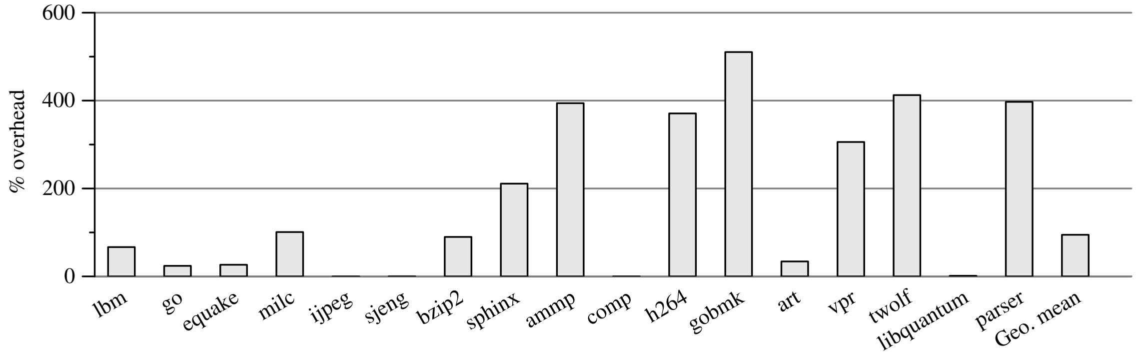


Figure 4.26: Memory overheads with *SoftBoundCETS*.

4.4.9 Instruction Overheads with *SoftBoundCETS*

Figure 4.25 reports the dynamic instruction count overheads with *SoftBoundCETS* instrumentation. On average, *SoftBoundCETS* instrumentation increases the dynamic instrumentation count by 226%. In most cases, this large number of instructions is the primary source of performance overhead. The dynamic instruction count overheads are highly correlated with the execution time performance overheads reported in Figure 4.19. The dynamic instruction count overhead is generally much larger than the corresponding execution time performance overheads. The Intel Core i7 processor we used for these experiments is a dynamically scheduled processor with a large instruction window and sophisticated branch predictor that can execute up to six micro-operations per cycle. As few programs have enough ILP to sustain six-wide execution, some of the instructions added by SoftBoundCETs are executed "for free" by the unused execution capacity. For example, benchmark *lbm* has 75% dynamic instruction count overhead but suffers just 9% execution time runtime overhead with *SoftBoundCETS*.

4.4.10 Memory Overheads with *SoftBoundCETS*

One cost of *SoftBoundCETS* instrumentation is that its disjoint metadata, shadow stack, and lock locations can increase the program's memory footprint. Figure 4.26 shows the normalized memory overheads based on the total number of 4KB pages touched during the entire execution with *SoftBoundCETS* instrumentation. We report the memory overheads for the benchmarks that we were able to measure with our infrastructure. For programs with many linked data structures (and thus many in-memory pointers), the worst-case memory footprint overhead can be high as shown in Figure 4.26. However, the memory overhead for many benchmarks is much lower. The average memory overheads are 94% on an average for our benchmarks.

4.5 Summary

This chapter presented *SoftBoundCETS* instrumentation on the LLVM IR. The presence of type information in the LLVM IR enabled *SoftBoundCETS* instrumentation to identify pointers for free and implement a pointer-based checking approach with disjoint metadata within the compiler. The presence of pointer information in the IR, the ability to instrument optimized code and the use of existing analyses enabled *SoftBoundCETS* instrumentation to attain low overheads without performing many custom analyses and optimizations. Further, simplicity of the instrumentation enabled us to provide memory safety even with separate compilation.

We organized the metadata as a trie data structure that enabled us to maintain pointer metadata in a disjoint metadata space without any operating system changes. Further, the use of shadow stack enabled *SoftBoundCETS* instrumentation to detect memory safety errors even when the callsite and the callee disagree on the function signatures. Beyond the advantages of *SoftBoundCETS* instrumentation, one primary limitation of *SoftBoundCETS* is that it disallows the creation of pointers from integers.

Chapter 5

Hardware Enforced Memory Safety

This chapter describes the implementation of the pointer-based approach with disjoint metadata purely within hardware. Unlike the implementations of the pointer-based approach described in Chapter 4 using the compiler, and that will be described in Chapter 6 using new hardware instructions with compiler support, this chapter describes an implementation purely in hardware that operates with mostly unmodified binaries with minimal changes to the tool chain. An alternative approach for retrofitting memory safety on unmodified binaries would have used either dynamic binary translation or dynamic binary instrumentation. As any extra instruction added during binary translation and binary instrumentation results in significant overheads, we propose pointer-based checking in hardware using micro-operation injection. We use the name *Watchdog* for our hardware instrumentation as it is the generalized version of our work on enforcing temporal safety in hardware [91]. Such an approach is attractive for hardware manufacturers as legacy programs can be retroffited with memory safety without requiring recompilation.

We initially describe the basic approach for adding instrumentation in the hardware to provide memory safety in Section 5.1. We discuss mechanisms for identifying pointer loads and stores to minimize metadata accesses in Section 5.2. We discuss optimizations to the pointer metadata propagation for register operations in Section 5.3. The summary of hardware changes is provided in Section 5.4. We evaluate the performance overheads to provide spatial and temporal safety in Section 5.5.

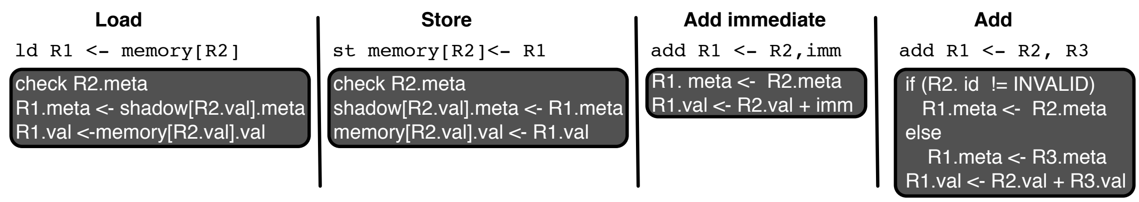


Figure 5.1: Metadata checking and propagation through *load*, *store*, *add-immediate* and *add*.

5.1 The *Watchdog* Approach

The goal of *Watchdog* is to implement pointer-based checking with disjoint metadata described in Chapter 3 using hardware instrumentation to provide *comprehensive detection*—detect all spatial and temporal errors—while keeping the overheads of checking every memory access low enough to be widely deployed in live systems for mostly unmodified binaries. *Watchdog* implements pointer-based checking by performing all the checking and propagation almost entirely in hardware, relying on the software runtime only to provide information about memory allocations and deallocations. To localize the hardware changes, this checking is implemented by augmenting instruction execution by injecting extra micro-operations (*uops*) [37]. Injecting *uops* keeps the hardware changes localized to the processor's front-end with rest of the processor pipeline being unmodified. Furthermore, *Watchdog* aims to attain the goals of source compatibility (*i.e.*, few source code changes) and binary compatibility (*i.e.*, library interfaces unchanged) by leaving the data layout unchanged using a disjoint shadow space for the metadata as described in Chapter 3.

5.1.1 Operation Overview

The *Watchdog* hardware is responsible for both metadata propagation and checking, and relies on the runtime to provide information about memory allocations and deallocations. Once the memory allocations are identified with the help of a modified *malloc()* and *free()* runtime library, a unique identifier (a lock and a key) is provided to every memory allocation and associated with the pointer pointing to the allocated memory along with the bounds information. As pointers can be resident in any register, conceptually *Watchdog* extends every register with a sidecar *metadata register*. Pointers can also reside in memory, so *Watchdog* provides a shadow memory that shadows every word of memory with identifier and bounds metadata for pointers. To propagate and check the metadata, *Watchdog* injects *uops* in hardware. On memory deallocations, the identifier associated

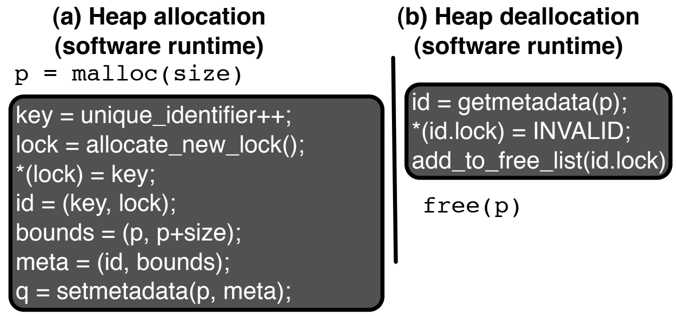


Figure 5.2: Metadata allocation and deallocation by the runtime with malloc/free and interfacing with the hardware using *setmetadata* and *getmetadata* instructions.

---

with the pointer pointing to the memory being deallocated is marked as invalid. On every memory access, *Watchdog* performs two checks: (1) a spatial check to ascertain that the pointer is in bounds and (2) a temporal check to ascertain if the identifier associated with the pointer being dereferenced is still valid by performing lock and key checking described in Chapter 3. Accessing a memory location either through a out-of-bounds pointer or using a pointer with an invalid identifier results in an exception. The following subsections explain each of these operations performed by *Watchdog*.

5.1.2 Metadata Assignment on Memory Allocation/Deallocation

To enforce bounds precisely, the hardware relies on byte-granularity bounds information provided by the compiler and/or runtime whenever a pointer is created [43]. For heap allocated objects, the *malloc()* runtime library can convey such bounds information. For pointers to a stack-allocated or global object, precise checking requires the compiler to insert instructions to convey bounds information at pointer creation points as described in Chapter 4. In the absence of such exact information, the hardware can still perform bounds checking — just less precisely — by restricting the bounds of pointers pointing to stack variables and globals to the range of the current stack frame and the global segment, respectively.

To provide temporal safety, identifier metadata must be allocated on memory allocations and invalidated on memory deallocations. Memory allocation/deallocation occurs when (1) the runtime performs such operations on the heap and (2) new stack frames are created/deleted on function entry and exits. Correspondingly, *Watchdog* allocates/deallocates identifier metadata on these operations. Heap allocation is fairly uncommon compared to function calls, so *Watchdog* relies on the runtime software to perform identifier management for the heap. In contrast, the *Watchdog* hardware performs the identifier management for function calls/returns.

On each heap memory allocation, the software runtime allocates both a unique 64-bit key and a new lock location from a list of free locations, and the runtime writes the key value into the lock location. The runtime conveys the identifier to the hardware using the *setmetadata* instruction that takes two register inputs: (1) a pointer to the start of the memory being allocated and (2) the 256 bit metadata (128-bit base/bound and 128-bit unique lock and key identifier) being assigned as shown in Figure 5.2(a). On memory deallocations, the runtime obtains the identifier associated with the pointer being freed using the *getmetadata* instruction that takes the pointer being freed as the register input as shown in Figure 5.2(b). The runtime then uses the identifier metadata to write an `INVALID` value to the lock location. The runtime then returns the lock location to the free list. To prevent double-frees and calling *free()* on memory not allocated with *malloc()*, the runtime also checks that the pointer's identifier is valid as part of the *free()* operation.

To perform identifier management for stack frames on calls and returns, the hardware injects *uops* to maintain an in-memory stack of lock locations whose top of stack is stored in a new *stack_lock* control register. The next key to be allocated is maintained with a separate *stack_key* control register. The base and bound for the stack pointer is set to be equal to the frame pointer and the frame pointer plus the maximum frame size respectively. On a function call, the hardware injects five *uops* to: allocate a new key, push that key onto the in-memory lock location stack, associate the new key and lock location with the stack pointer, and associate the bounds with the stack pointer (see Figure 5.3(a)). On function return, the identifier associated with the stack pointer is restored to the identifier of the current stack frame. This operation is accomplished by reading the value of the key from the memory location pointed by the stack lock register after the stack manipulation. Similarly the bounds of the stack pointer is set to be equal to the bounds of the current stack frame (for a total of five *uops*, as shown in Figure 5.3(b)).

5.1.3 Checks on Memory Accesses

The hardware performs spatial and temporal checks using the metadata in the sidecar register conceptually before every memory access. The *check uop* uses the sidecar metadata associated with the pointer register being dereferenced and performs (1) the lock and key checking and (2) bounds

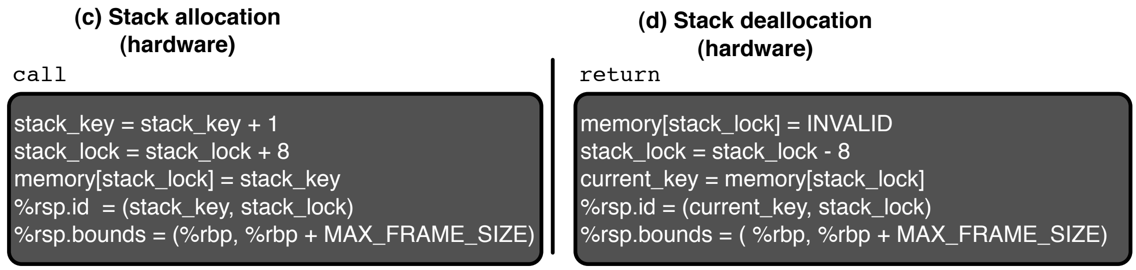


Figure 5.3: Metadata allocation and deallocation with call/return

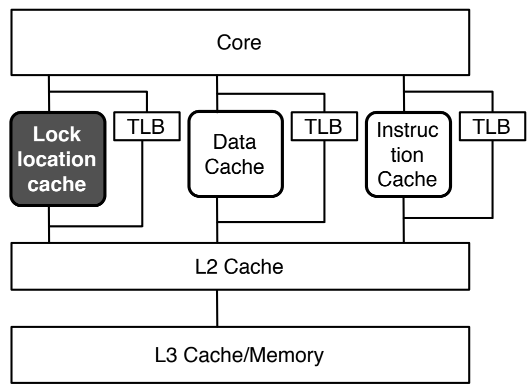


Figure 5.4: Placement of the lock location cache (shaded).

checking described in Chapter 3. The bounds check compares the pointer with the sidecar bounds metadata. The temporal check compares the key in the sidecar identifier metadata with the key at the memory location pointed by the lock part of the sidecar identifier metadata. A check failure triggers an exception, which can be handled by the operating system by aborting the program or by invoking some user-level exception handling mechanism.

There are two alternatives for performing checks, and we explore both the alternatives. In the first alternative, the hardware performs both the checks based on the metadata by injecting two *uops*— a bounds check *uop* and a temporal check *uop*— for each memory operation. In the second alternative implementation, the hardware injects a single check *uop* to perform both checks in parallel. As the bounds check consists of just two inequality comparisons (and requires no additional memory accesses), either implementation is likely feasible. We evaluate both alternatives in Section 5.5.

In contrast to bounds check where all inputs are registers, the temporal check performs a memory access and a equality comparison, which increases the demand placed on the cache ports. To mitigate this impact, *Watchdog* optionally adds a *lock location cache* to the core, which is accessed by the *check uop* and is dedicated exclusively for lock locations. Just as splitting the instruction and data caches increases the effective cache bandwidth (by separating instruction fetches from loads/stores), this additional cache is used to provide more bandwidth for accessing lock locations. This cache becomes a peer with the instruction and data caches (as shown in Figure 5.4), has its own (small) TLB, and uses the same tagging, block size, and state bits used to maintain coherence among the caches. Memory allocations and deallocations update lock location values, so these operations also access the lock location cache. Even a small lock location cache (*e.g.*, 4KB) can be effective because (1) lock locations (8 bytes per object currently allocated) are small relative to the average object size and (2) the lock locations region has little fragmentation and exhibits reasonable spatial locality because lock locations are reallocated using a LIFO free list. Cache misses are handled just like misses in the data cache.

5.1.4 In-Memory Pointer Metadata

As pointers can be resident in memory, the metadata also needs to be maintained with pointers in memory. To maintain memory layout compatibility, the hardware maintains the per-pointer metadata in the shadow memory. Conceptually, every word in memory has bounds and identifier metadata in the shadow memory. When a pointer is read from memory, the metadata associated with the pointer being read is also read from the shadow memory. To implement this behavior (see Figure 5.1(a)), for every load instruction the *Watchdog* hardware injects (1) a `check` *uop* to perform the check, (2) a *uop* to perform the load of the actual value into the register and (3) a `shadow_load` *uop* to load the metadata (meta) from the shadow memory space. Stores are handled analogously (also shown in Figure 5.1(b)). We assume pointers are word aligned (as is required by some ISAs and is generally true with modern compilers even for x86), which allows the the shadow load/store *uops* to accesses the shadow space via an aligned load/store in a single cache access.

The shadow space is placed in a dedicated region of the virtual address space that mirrors the normal data space. Placing the shadow space into the program's virtual address space allows shadow accesses to be handled as normal memory accesses using the usual address translation and page allocation mechanisms of the operating system. Current 64-bit x86 systems support 48-bit virtual addresses, so the the hardware uses a few high-order bits from the available virtual address space to position the shadow space. This organization allows the shadow_load/shadow_store *uop* to convert an address to a shadowspace address via simple bit selection and concatenation.

Accessing the shadow space on every memory operation would result in significant performance penalties, so Section 5.2 describes the mechanisms we use to reduce the number of metadata accesses by inserting metadata load/store *uops* only for those memory operations that might actually load or store pointer values.

5.1.5 In-Register Metadata

To ensure that the checks inserted before a memory access have the correct metadata, the bounds and identifier metadata must be propagated with all pointer operations acting on values in registers (pointer copies and pointer arithmetic). For example, when an offset is added or subtracted from a pointer, the destination register inherits the metadata of the original pointer. Figure 5.1 shows the bounds and identifier metadata propagation with addition operations as a result of pointer arithmetic. Unlike the compiler instrumentation described in Chapter 4 that performed metadata propagation for pointers in temporaries by either creating temporaries or copy-propagation in the intermediate representation, *Watchdog* performs it by injecting extra propagation *uops* as shown in Figure 5.1. Such register manipulation instructions are extremely common, so copying the metadata on each operation (say, via an inserted *uop*) would be extremely costly. Instead, Section 5.3 describes *Watchdog*'s use of copy elimination via register renaming to reduce the number of propagation *uops* inserted.

This section has described the basic approach for implementing pointer-based checking with disjoint metadata in hardware for almost unmodified binaries and has outlined two implementation optimizations to make it efficient: identifying pointer accesses and register renaming techniques to avoid unnecessary *uops*. The next two sections describe these design optimizations, respectively.

5.2 Identifying Pointer Load/Store Operations

Watchdog maintains bounds and identifier metadata with every pointer in a register or in memory. However, binaries for standard ISAs do not provide explicit information about which operations

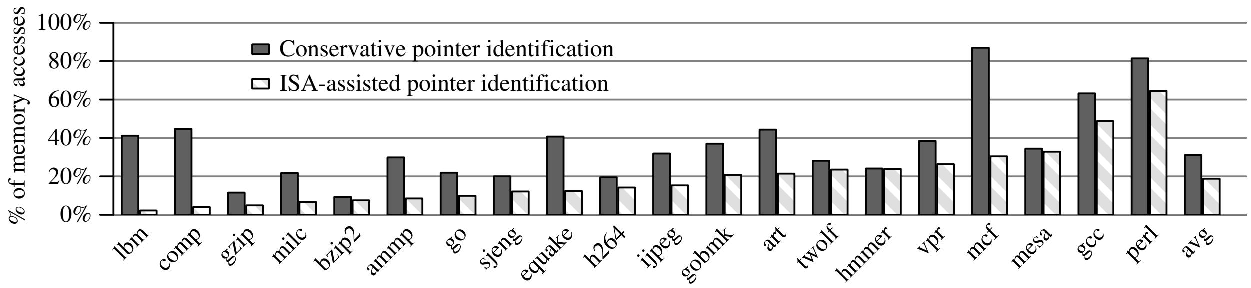


Figure 5.5: Percentage of memory accesses metadata for conservative and ISA-assisted identification.

manipulate pointers. In the absence of such information, propagating metadata with every register and memory operation would require many extra memory operations, resulting in substantial performance degradation. This section describes two techniques for identifying pointer operations, the results of which can be used to reduce the number of accesses to the metadata space.

5.2.1 Conservative Pointer Identification

To enable *Watchdog* to work with reasonable overhead without significant changes to program binaries, we observe that, for current ISAs and compilers, pointers are generally word-sized, aligned, and resident in integer registers. Based on this observation, *Watchdog* conservatively assumes that only a 64-bit load/store to an integer register may be a pointer operation, whereas floating-point load/stores and sub-word memory accesses are non-pointer operations. *Watchdog* does not insert additional metadata manipulation *uops* for such non-pointer operations.[1] As evidence of the effectiveness of this heuristic, the left bars in Figure 5.5 show that this approach classifies 31% of memory operations as potentially loading/storing a pointer.

5.2.2 ISA-Assisted Pointer Identification

Although the above conservative heuristic is effective and requires no ISA modifications, we can be more precise if the ISA is modified to allow the compiler to inform the hardware of exactly which instructions are manipulating pointers. There are two primary ways *Watchdog* could get information about memory accesses that load/store a pointer.

[1]One potentially problematic case is the manipulation of pointers using byte-by-byte copies (*e.g.* `memcpy()`). We found that compilers for x86 typically use word-granularity operations. In other cases, we have modified the standard library as needed.

First, the registers in the ISA could be separated into pointer registers and normal data registers. With such an ISA, *Watchdog* would need to perform work only on operations concerned with pointer registers. Accesses to the metadata space need to be introduced only when pointer operations are loaded or stored. However, this would requires significant changes to the ISA, making it perhaps more suitable only when designing a new ISA.

Second, the ISA could be extended to include variants of loads and stores that indicate a pointer is being operated on. The compiler, which generally knows which operations are manipulating pointers, would then select the proper load/store variant. For any statically ambiguous case, the compiler would conservatively select the memory operation variant that performs the metadata operations.

To study the benefits of either approach — but without designing a new ISA or performing significant modifications to a compiler backend and ISA encoding — we explore the potential for such ISA extension by profiling the set of static instructions that ever loaded/stored valid pointer metadata. To generate such profiles, we configured *Watchdog* to access the metadata space on every memory operation and identified PCs which ever loaded valid metadata. For subsequent runs, we consider these static memory operations as having been marked by the compiler as load/stores of pointers.[2]

The rightmost bar of each benchmark in Figure 5.5 shows the percentage of memory access that are classified as pointer operations using this approach. This reduces the number of memory accesses classified as pointer operations from 31% with conservative pointer identification to 18%. For benchmarks like `compress` and `lbm`, the improvement is significant as almost every memory access is classified as non-pointer memory access.

The number of pointer loads and stores reported in Figure 5.5 for ISA-assisted pointer identification is slightly higher than number of pointer loads/stores identified using the type information in the LLVM IR in Chapter 4. This artifact is attributed to *Watchdog*'s pointer load/store identification scheme. A memory access instruction at a particular PC that loads/stores pointers on some accesses and non-pointer values on others are classified as requiring a metadata access all the time. Hence, the estimates of metadata access elimination is conservative compared to what one would obtain with a explicit pointer register or a mode where compiler indicates that pointer is being loaded/s-

[2]Although we see this approximation as primarily an experimental aide, such an approach might actually be useful for instrumenting libraries or other code that cannot be recompiled.

tored. One concrete instance of such a PC that loads/stores pointers sometimes and non-pointers otherwise, occurs when the compiler spills and restores values on the stack. When pointers are spilled, metadata accesses need to be performed which are not necessary otherwise. A compiler that has precise information about the pointer spills can provide information and eliminate the unnecessary metadata accesses.

5.3 Decoupled Register Metadata

As described thus far, *Watchdog* explicitly copies metadata along with each arithmetic operation in registers. This section briefly discusses and discards the straightforward approach of widening each register with additional metadata. Although such a design might be appropriate for an in-order processor core, our *Watchdog* implementation targets out-of-order dynamically scheduled cores. Thus, this section: (1) describes a decoupled metadata implementation of *Watchdog* in which the data and metadata are mapped to different physical registers within the core, (2) discusses what *uops* would be inserted to maintain the decoupled register metadata, and (3) shows how metadata propagation overheads can be reduced via previously-proposed copy elimination modifications to the register renaming logic.

5.3.1 Strawman: Monolithic Register Data/Metadata

As presented in Section 5.1, *Watchdog* views each register as being widened with a sidecar to contain identifier metadata. Although that design is conceptually straightforward—especially for an in-order core—it suffers from inefficiencies. First, every register write (and most register reads) must access the sidecar metadata, increasing the number of bits read and written to the register file; although not necessarily a performance problem, this could be a energy concern. Second, a more subtle issue is that treating register data/metadata as monolithic causes operations that write just the data or metadata to become partial register writes. These partial register accesses introduce unnecessary dependencies between *uops*, for example, the load *uop* and the metadata load *uop*. These serializations can have several detrimental effects, including (1) increasing the load-to-use penalty of pointer loads, (2) stalling subsequent instructions if either of the loads miss in the cache, and (3) limiting the memory-level parallelism by serializing the load and the metadata load. In our initial experiments with various implementations of monolithic registers, we found the performance impact of such serializations to be significant.

5.3.2 Decoupled Register Data/Metadata

To address these performance issues, *Watchdog* decouples the register metadata by maintaining the data and metadata in separate physical registers. Each architectural register is mapped to two physical registers: one for data and one for the metadata. With this change, individual *uops* generally operate on either the data or the metadata, removing the serialization caused by partial register writes of monolithic registers. Once decoupled, the metadata propagation and checking is almost entirely removed from the critical path of the program's dataflow graph.

With decoupled metadata, there are multiple cases for which register metadata must be propagated or updated. First, instructions such as adding an immediate value to a register simply copy the metadata from the input register to the output register. Second, some instructions never generate valid pointers (*e.g.*, the output of a sub-word operation or a divide is not a valid pointer), thus such instructions always set the metadata of the output register to be invalid. Third, either of the registers might be a pointer, so for such instructions *Watchdog* inserts a *select uop*, which selects the metadata from whichever register has valid metadata.[3]

Watchdog performs metadata propagation by changing the register renaming to reduce the number of extra *uops* inserted. In only one of these three cases described above does *Watchdog* actually insert *uops*; in the other cases (copying the metadata or setting it to invalid), *Watchdog* uses previously proposed modifications to register renaming logic [76, 106] to handle these operations completely in the register rename stage. *Watchdog* extends the maptable to maintain two mappings for each logical register: the regular mapping and a metadata mapping. Instructions that unambiguously copy the metadata (such as "add immediate", which has a single register input) update the metadata mapping of the destination register in the maptable with the metadata mapping entry of the input register. This implementation eliminates the register copies by physical register sharing, as there is a single copy of the metadata in a physical register [106]. To ensure that this physical register is not freed until all the mappings in the maptable are overwritten, these physical registers

[3]If the ISA was further extended to allow the compiler to annotate such instructions, these select *uops* could also be eliminated.

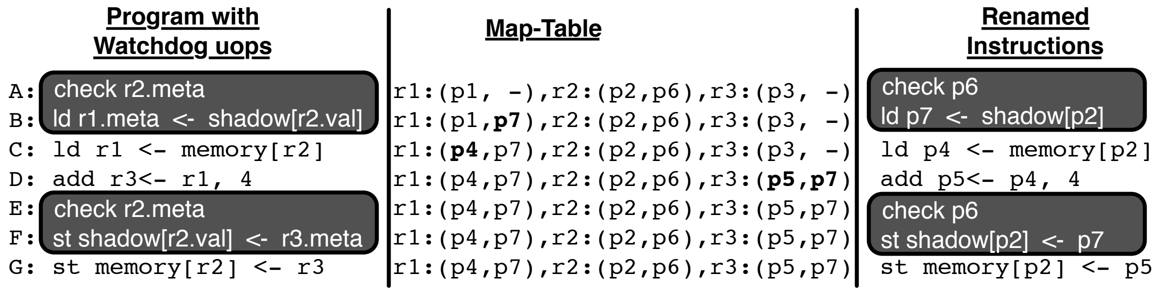


Figure 5.6: Example illustrating register renaming with *Watchdog uops* and extensions to the map table. Watchdog inserted *uops* are shaded. The map table is represented by a tuple for each register. r:(a,b) means logical register r maps to physical register a according to the regular map table mapping and the logical register r maps to a 256-bit physical register b according to the *Watchdog* mapping. *Watchdog* introduced load and store *uops* access the shadow memory (`shadow`) for accessing the metadata. The watchdog mapping of $-$ indicates the invalid mapping (the register currently contains a non-pointer value).

---

need to be reference counted. We adopt previously proposed techniques to efficiently implement reference counted physical registers [112].

5.3.3 Decoupled Metadata Example

Figure 5.6 illustrates the decoupled register metadata and operation of *Watchdog* with extensions to the map table. The map table for each architectural register is a tuple containing a regular mapping and a *Watchdog* mapping to physical registers. The *Watchdog* inserted *uops* are shaded and the state of the map table after renaming each *uop* is shown in the figure. Initially registers $r1$ and $r3$ have invalid mappings for their *Watchdog* mapping.

The check uop instruction (A) introduced before the load from register $r2$ uses the physical register $p6$, the watchdog mapping for register $r2$. Instruction (B), the *Watchdog* inserted uop to load the metadata from the metadata space uses the value of the register $r2$ as input and hence uses the regular mapping for the input register *i.e.* $p2$ and allocates a new physical register $p7$ and updates the *Watchdog* mapping for architectural register $r1$. Instruction (C), the load operation performs the normal operation by just updating the regular mappings of the map table.

The add instruction adds a constant to the pointer in architectural register $r1$ and writes it to architectural register $r3$. As this instruction writes a register, a new physical register is allocated and assigned to the regular mapping of architectural register $r3$. However, this instruction also

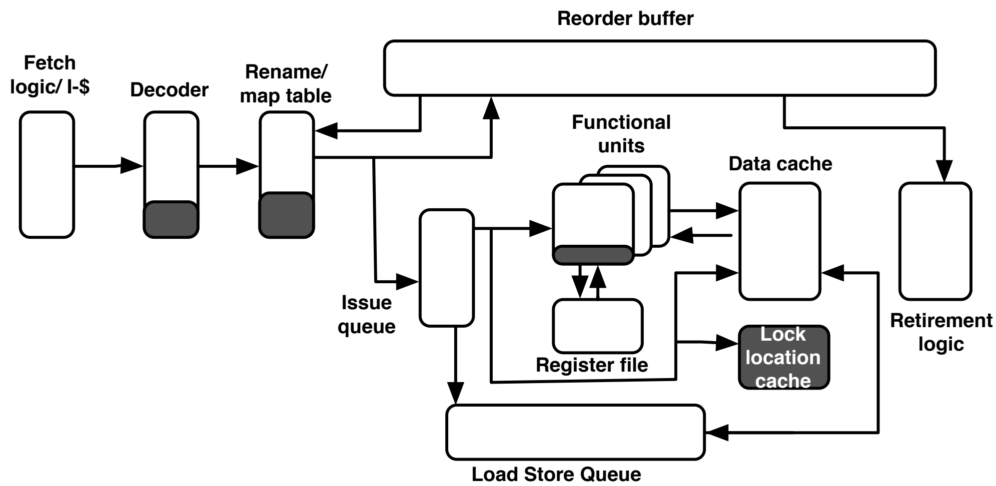


Figure 5.7: Illustration of changes to the processor core with *Watchdog* in comparison to a traditional out-of-order core. *Watchdog* specific modifications are shaded.

propagates pointer metadata, thus the *Watchdog* mapping of the architectural register $r3$ is updated to be the *Watchdog* mapping of the architectural register of input pointer register ie. $r1$. Thus, the *Watchdog* mapping of architectural register $r3$ is updated to be physical register $p7$ eliminating register copies.

To summarize, Figure 5.6 illustrates that decoupled metadata does not introduce unnecessary dependencies between *uops*. Hence, the decoupled register metadata with the map table modifications prevents the serialization that occurs with the straightforward monolithic register implementation, and it also avoids additional *uops* for metadata copies in most cases.

5.4 Summary of Hardware Changes

Figure 5.7 illustrates the changes to the traditional out-of-order processor core to perform pointer-based checking with disjoint metadata. *Watchdog* requires changes to the decoder and the instruction cracker to inject *uops* to perform metadata propagation and checking. We add two new instructions *setmetadata* and *getmetadata* to enable the runtime to communicate with *Watchdog*. The x86 ISA already supports decoding more than one thousand instructions. Adding additional two instructions does not add significant complexity. Further, *uop* injection can be performed at the end of regular instruction decode as a post processing step. As, these changes are localized, it will not add significant chip verification burden.

The second main modification with *Watchdog* concerns the register renaming logic in the out-of-order core as described in Section 5.3. There are two changes: (1) the map table is extended with another field for *Watchdog* mappings, and (2) the register renaming logic is changed to update *Watchdog* specific metadata mappings. We estimate the cost of adding such modifications to an x86-like out-of-order core. First, *Watchdog* mapping extension to the map table adds 9-bits per entry of the map table (assuming that the machine has a maximum of 512 physical registers). There is a maptable entry for each logical register. There are a total of 36 entries in the map table (sixteen regular registers + sixteen SIMD registers + four temporary registers). The additional bits added to the renaming logic is 324-bits (9*36) which is approximately 41 bytes. Thus, the additional logic required to implement these modifications are small. Further, we also need to add reference counts to support physical register sharing that adds a few additional bits to each physical register.

The other modifications to the core include the addition of a lock location cache, and additional logic with the functional units as shown in Figure 5.7. The lock location cache added to mitigate the contention on the ports adds 4KB to the area overhead. These hardware extensions proposed to perform pointer-based checking are either comparable or less invasive compared to prior hardware approaches that provide partial safety [29, 32, 43]. We further reduce these hardware overheads with the help of the compiler in our *HCH* instrumentation that will be described in Chapter 6.

5.5 Performance Evaluation

This section provides an experimental evaluation of *Watchdog* while: (1) highlighting its low performance overheads, and (2) providing an understanding of the contribution of various techniques proposed in this chapter in reducing the performance overheads.

5.5.1 Methodology

This subsection describes the methodology adopted to measure the performance overheads. We describe the simulator, benchmarks and the sampling techniques used.

Simulator To evaluate the benefits of hardware extensions, we used an x86-64 simulator which executes the user-level portions of statically linked 64-bit x86 programs. The simulator was originally developed by Andrew Hilton as part of his dissertation research on latency tolerant processors [68, 69, 70]. We have further extended the simulator extensively and added features to simulate a wide range of x86 binaries. The simulator decodes x86 macro instructions and cracks them into a RISC-style *uop* ISA. The *uop* cracking used in Core i7 "Sandy Bridge" processor is proprietary and not publicly available, so there is no way to tell how similar the simulator's *uop* ISA is to what is actually used by Intel. Table 5.1 describes the configurations used for each component of the micro-architecture. The out-of-order processor configurations described are designed to be similar to Intel's Core i7 "Sandy Bridge" processor. We model the details of Core i7 using the publicly available information such as the memory hierarchy (large L3 cache split into banks on a ring interconnect with private L2/L1), and structure sizes (ROB, LQ, SQ, IQ, etc). We simulate a three level cache hierarchy with private L1 and L2 caches of size 32KB and 256KB respectively and a shared L3 cache of size 16MB divided into 4 banks organized as a ring. We modified the standard DLmalloc memory allocator to use the new instruction to inform the hardware of memory allocations and deallocations.

Benchmarks We used twenty C SPEC benchmarks from the SPEC2006, SPEC2000, and SPEC95 benchmark suites. We compiled the benchmarks using the GNU C compiler version 4.4 using standard optimization flags. We generally used the reference inputs, but used train/test inputs in some cases to ensure reasonable simulation times. We used 2% periodic sampling with each sample of 10 million instructions proceeded by a fast forward and a warmup of 480 and 10 million instructions per period, respectively. We use execution time, which is calculated using the macro instruction IPC (Instructions per cycle) and the number of instructions executed, to measure the performance overhead.

5.5.2 Runtime Overheads of *Watchdog*

Figure 5.8 presents the percentage execution time overhead of *Watchdog* over a baseline without any *Watchdog* instrumentation (smaller bars are better as they represent lower runtime overheads). The graphs contains a pair of bars for each benchmark. The height of the left and right bars represent the overhead of *Watchdog* with conservative pointer identification (38% on average) and ISA-assisted

	Clock	3.4 GHz
Front-end	Bpred	3-table PPM: 256x2, 128x4, 128x4, 8-bit tags,2-bit counters
	Fetch	16 bytes/cycle. 3 cycle latency
	Rename	Max 6*uops* per cycle. 2 cycle latency
	Dispatch	Max 6*uops* per cycle. 1 cycle latency
Window/Exec	Registers	(160 int + 144 floating point), 2 cycle
	ROB/IQ	168-entry ROB, 54-entry IQ
	Issue	6-wide. Speculative wakeup.
	Int FUs	6 ALU. 1 branch. 2 ld. 1 st. 2 mul/div
	FP FUs	2 ALU/convert. 1 mul. 1 mul/div/sqrt.
	LQ size	64-entry LQ
	SQ size	36-entry SQ
Memory Hierarchy	L1 I$ Prefetcher	32KB. 4-way, 64B blocks. 3 cycles 2-streams, 4 blocks each
	L1 D$ Prefetcher	32KB, 8-way, 64B blocks, 3 cycles 4-streams, 4 blocks each
	L1 $\leftrightarrow$ L2 bus	32-bytes/cycle. 1 cycle.
	Private L2$ Prefetcher	256KB, 8-way, 64B blocks, 10 cycles. 8 streams. 16 blocks.
	L2 $\leftrightarrow$ L3 bus	8-stop bi-directional ring. 8-bytes/cycle/hop. 2.0GHz clock
	Shared L3$	16MB. 16-way, 64B blocks, 25 cycles
	Mem. Bus	800MHz. DDR. 8-bytes wide. Dual channel. 16ns latency
	Lock Location $	4KB, 8-way, 64B blocks

Table 5.1: Simulated processor configurations

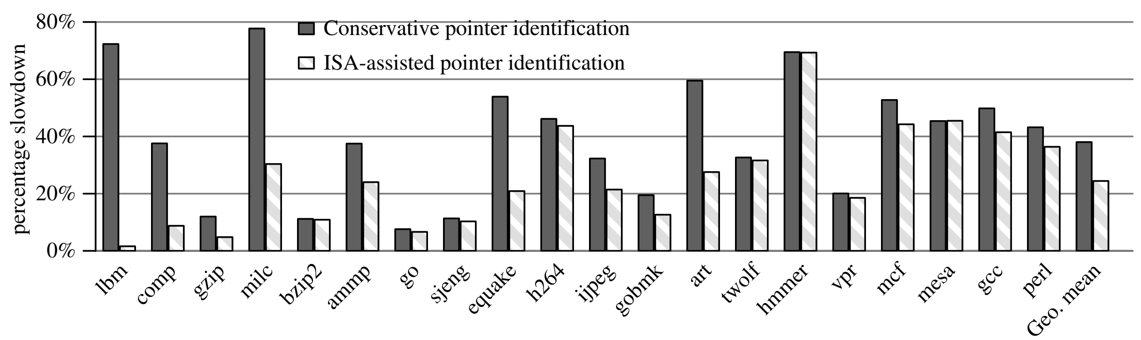


Figure 5.8: Execution time performance overhead of Watchdog with conservative and ISA-assisted pointer identification.

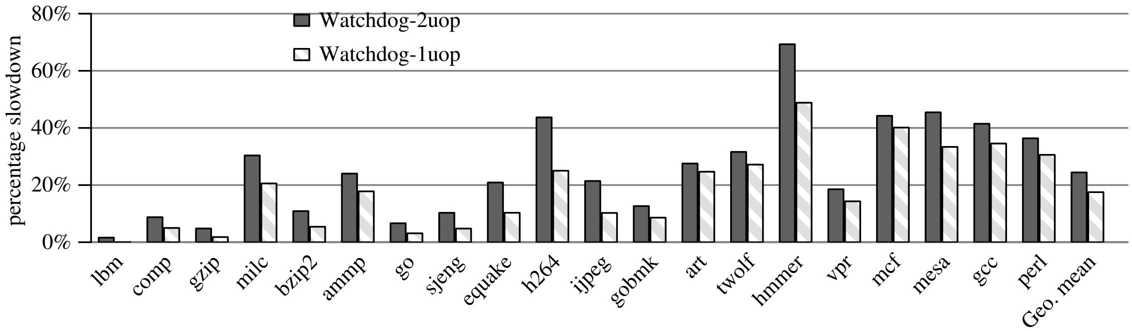


Figure 5.9: Watchdog's overhead with a single check uop and two check uops with ISA-assisted pointer identification.

---

pointer identification (24% on average), respectively. These runtime are substantially lower than the overhead reported by related compiler instrumentation described in Chapter 4. ISA-assisted pointer identification reduces the performance overhead for enforcing full memory safety significantly for a large number of benchmarks. For example, for benchmark *lbm*, ISA-assisted pointer identification reduces the performance overhead from 72% with conservative identification to just 2%, which correlates with the reduction in the number of pointer loads and stores in Figure 5.5 on page 100.

Impact of Separate Check *uops* on Performance *Watchdog* has two alternatives to perform both spatial and temporal checks: (1) inject a pair of *uops* (one *uop* each for spatial check and a temporal check) or (2) inject a single check *uop* that performs both spatial and temporal checks. When implemented as a single *uop*, the overall number of *uops* is reduced by the number of loads and stores. Figure 5.9 reports the performance overhead of *Watchdog* with two check *uops* and a single check *uop*. Using a single check *uop* results in a 7% decrease in execution time overhead (from 24% to 17% on average). Benchmarks like hmmer, h264 and milc that have high IPC (instructions per cycle) and saturate the execution resources benefit from a reduction in the number of check *uops*.

***uop* Overheads** *Watchdog* performs its functionality by inserting *uops*, so the total number of *uops* inserted is instructive in understanding the sources of execution time overheads. Figure 5.10 presents the *uop* overhead when employing ISA-assisted pointer identification. The total height of the each bar represents the total *uop* overhead for the benchmark and each bar is divided into four segments: (1) checks, (2) pointer loads, (3) pointer stores, and (4) the *uops* to perform memory

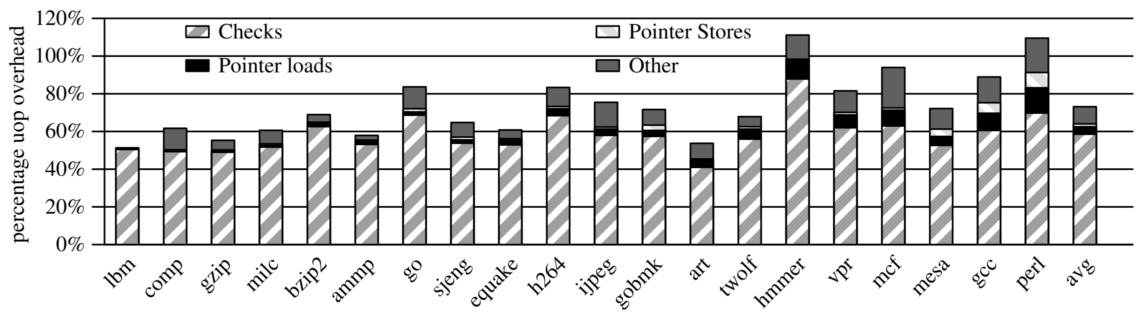


Figure 5.10: *Watchdog*'s uop overhead.

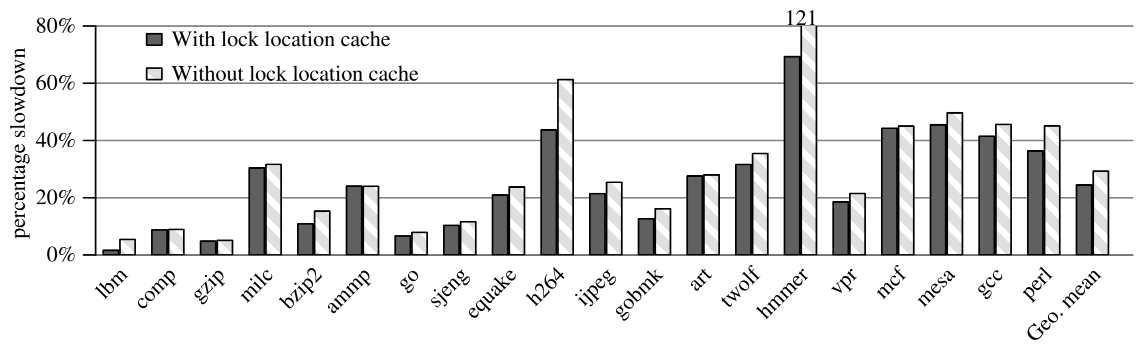


Figure 5.11: Watchdog's overhead with and without a lock location cache for ISA-assisted pointer identification.

allocation/deallocation and metadata propagation in registers. On average, *Watchdog* executes 73% more *uops* than the baseline. The execution time overhead is lower than the *uop* overhead because these *uops* are off the critical path and thus execute in parallel as part of superscalar execution. The check *uops* account for bulk of the *uop* overhead (58% on average: 29% each for spatial and temporal checks). Pointer metadata load and store *uops* account for 4% and 2% of the extra *uops* on an average but can be as high as 14% and 8%, respectively. The *uop* overhead due to propagation *uops* and memory allocation/deallocation operations (on the heap and the stack) account for the remaining *uops* (9% on average). *Watchdog* with a single check uop reduces the *uop* overhead from 73% to 44% on average.

Impact of Lock Location Cache on Performance To decrease contention on limited cache ports, the results presented thus far include a 4KB lock location cache (Figure 5.4 on page 97). Figure 5.11 reports the execution time overhead without this cache, in which all check operations use the lim-

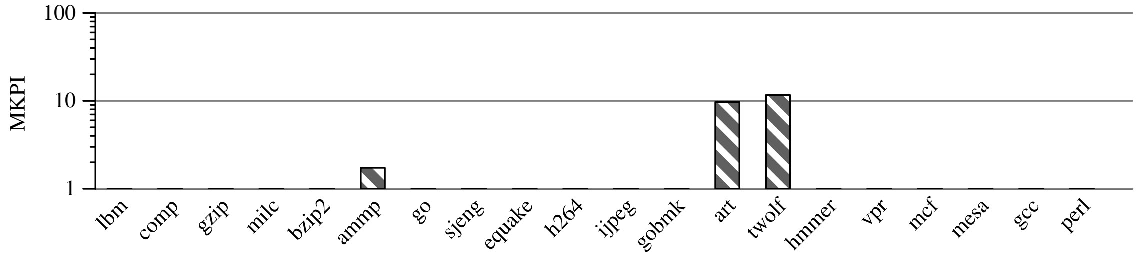


Figure 5.12: Misses per thousand instructions with a lock location cache of 4KB.

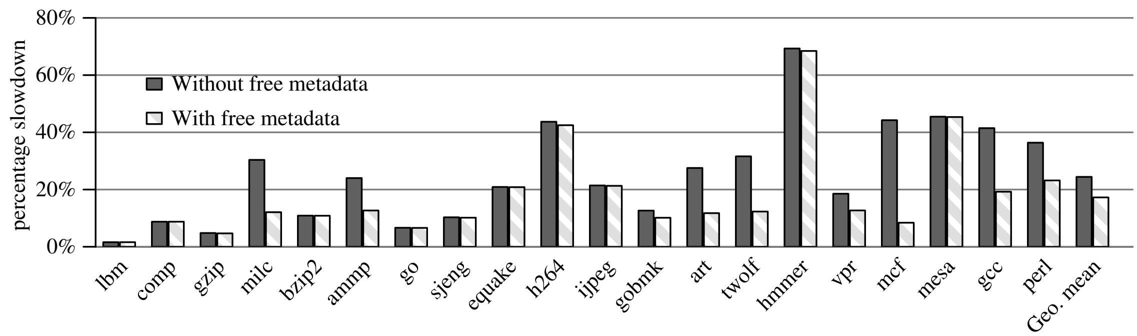


Figure 5.13: Watchdog's overhead when there are no metadata misses in the cache.

ited data load ports. Without the lock location cache, the overhead of *Watchdog* increases to 29% on average (up from 24%). The improvements are especially significant for benchmarks such as `hmmer` and `h264`, as these benchmarks already have high IPC and frequent memory accesses, both of which lead to significant contention for the data cache load ports in the absence of a lock location cache. These results are not particularly sensitive to the exact size of the lock location cache; for a 4KB cache, the miss rate is less than 1 miss per 1000 instructions for seventeen of the twenty benchmarks as shown in Figure 5.12.

Impact of Metadata on Cache Misses One potential source of performance overhead is the additional cache pressure due to the per-pointer shadowspace metadata. To isolate this effect, we performed a set of simulations configured to idealize the shadow memory accesses (metadata accesses occupy cache ports but never cache miss and do not actually consume space in the data cache). Making the metadata free of cache effects in this way changed the runtime overhead by only 7%

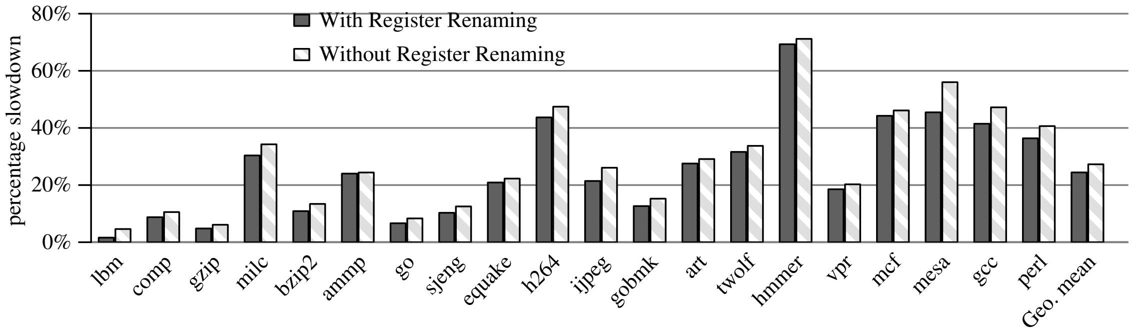


Figure 5.14: Watchdog's overhead when there are additional select uops without physical register sharing.

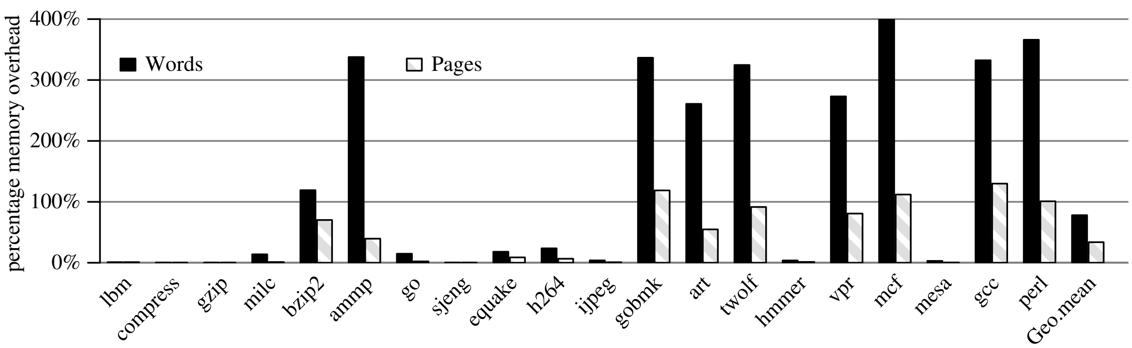


Figure 5.15: *Watchdog*'s memory overhead with words and pages.

on average (decrease from 24% to 17%), indicating that cache pressure effects are generally not dominant in these benchmarks as shown in Figure 5.13.

Impact of Physical Register Sharing on Performance. In the absence of precise information about operations producing pointer values, *Watchdog* introduces *select* uops to allocate a separate physical register to choose the metadata of the output register from the input registers as described in Section 5.3. Figure 5.14 reports the performance overhead when *select* uops are introduced for all integer instructions that produce a value instead of physical register sharing described in Section 5.3. On average, introduction of additional *select* uops increases the overhead from 24% to 27% on average. The performance overhead benefits from physical register sharing are low likely due to two reasons: (1) the wide execution configuration of our simulated machine, and (2) a single *select uop* inserted for propagating 256-bits of metadata.

Memory Overheads with *Watchdog*. Figure 5.15 presents the memory overheads with ISA-assisted pointer identification for providing both spatial and temporal safety over a baseline without any instrumentation. The memory overhead is negligible for the majority of the benchmarks. However, several of the benchmarks approach worst-case overheads of four shadow pages for each non-shadow page. The memory overhead is calculated in two ways: total words of memory accessed (left bar) and total 4KB pages of memory accessed (right bar), which reflects on-demand allocation of shadow space pages by the operating system. On average, the memory overhead calculated these two ways is 36% and 84%, respectively. The difference in these metrics reflect the impact of fragmentation caused by page-granularity allocation of the shadow space. These memory overheads compare favorably to the overheads reported for garbage collection [66] or prior best-effort approaches for mitigating use-after-free errors such as heap randomization [88] and object-per-page approaches [45]. Although the memory overhead adds to the system cost, the performance impacts of the additional cache pressure were already included in the performance results.

5.6 Summary

This chapter presented *Watchdog*, a hardware implementation of the pointer-based checking approach with disjoint metadata described in Chapter 3 on mostly unmodified binaries. The key challenge in implementing pointer-based checking on binaries is the absence of pointer information. Extending the tool chain to provide pointer information or identifying good heuristics to identify pointers can enable hardware to provide memory safety at reasonable overheads without sophisticated compiler instrumentation and analyses as demonstrated by our ISA-assisted pointer identification. We observed that preventing unnecessary dependences between the injected *uops* and original *uops* is crucial for performance. Further, avoiding structural hazards using the lock location cache and redundant copies using physical register sharing with *Watchdog* demonstrates that it is feasible to implement hardware-based pointer-based checking with disjoint metadata on mostly unmodified binaries.

Chapter 6

Hardware-Accelerated Compiler Instrumentation for Memory Safety

Over the last few decades, chip designers have introduced numerous ISA extensions for performing specific computations. The most notable example is the multiple generations of Streaming SIMD Extensions (SSE) with instructions specifically tailored for accelerating digital signal processing, video encoding, cyclic redundancy check (CRC) calculation, string parsing, and encryption. This chapter makes a case for *memory safety checking* as the next family of computations ripe for acceleration by adding new instructions.

Unlike the pointer-based checking approach purely either in the compiler (Chapter 4) or in hardware (Chapter 5), this chapter describes the new ISA extensions to provide hardware acceleration for the compiler instrumentation described in Chapter 4 for enforcing memory safety. We use the name *HCH*—Hardware Compiler Hybrid—for our hybrid approach with the new ISA extensions. In this hybrid approach, the compiler instrumentation performs metadata propagation and inserts the custom instructions provided by the hardware for metadata loads/stores, spatial checks, and temporal checks. The key goal of this hybrid approach is to accelerate the compiler instrumentation described in Chapter 4 while minimizing the hardware changes. We first revisit the sources of instruction overhead in compiler-based instrumentation described in Chapter 4 and explore the opportunities for hardware acceleration (Section 6.1). We describe the new hardware instructions to minimize few of the sources of overhead (Section 6.2). We describe extensions to the compiler instrumentation to use new vector extensions on modern processors and propose new vector in-

Spatial Check

```
// rax -- pointer
// rcx -- sizeoftype
// rdx -- base
// r15 -- bound

cmp rax, rdx
jb <abort_address>
lea rbx, [rax+rcx]
cmp rbx, r15
ja <abort_address>
```

(a)

Temporal Check

```
// rcx -- key
// rdx -- lock

mov rbx, [rdx]
cmp rbx, rcx
jne <abort_address>
```

(b)

Control Flow Check

```
// rax -- pointer
// rdx -- base
// r15 -- bound

cmp rax, rdx
jne <abort_address>
cmp rax, r15
jne <abort_address>
```

(c)

Figure 6.1: The x86 instructions inserted by the compiler instrumentation for various checks.

structions to mitigate some of the overheads (Section 6.4). We provide a qualitative comparison of *HCH* and *Watchdog* (Section 6.6). We evaluate the overheads of the compiler instrumentation in the presence of new hardware instructions (Section 6.8).

6.1 Sources of Instruction Overhead with Compiler Instrumentation

The extra instrumentation code added by the compiler instrumentation described in Chapter 4 results in instruction overhead. These instruction overheads likely introduce performance overheads when the program is executed. The main sources of overhead with the compiler instrumentation are attributed to the following: (1) checks, (2) metadata loads/stores, (3) shadow stack operations, and (4) second-order effects such as extra spills and restores.

There are three main types of checks introduced by *SoftBoundCETS*. First, spatial checks are inserted conceptually before every memory access. Spatial checks introduce both extra control flow and instructions. In our compiler prototype a spatial check is five extra x86 instructions as shown in Figure 6.1(a). The temporal check conceptually before every memory access also introduces extra instructions. Further, a temporal check performs a memory access to load the key at the lock location as described in Chapter 3 introducing pressure on the cache ports in the machine. In our compiler prototype, a temporal check is three x86 instructions as shown in Figure 6.1(b), which will likely be fused with macro-operation fusion on modern Intel processors. The control flow check introduced to check indirect function calls is another source of check overhead, which is approximately four x86 instructions in our prototype as shown in Figure 6.1(c).

Metadata Load

Inputs: %rdi - address_of_ptr

Outputs: %r12 - base   %r13 - bound

%r14 - key   %r15 - lock

```
movq %rax, %rdi // %rax  = %rdi
shrq %rax, $22
movabsq %rcx, $4398046511096
andq %rcx, %rax
leaq %rax, (,%rdi,4)
andq $134217696, %rax
movq %rdx, _trieroot(%rip)
addq %rax, (%rdx, %rcx)

movq %r12, (%rax)   // load base
movq %r13, 8(%rax)  // load bound
movq %r14, 16(%rax) // load key
movq %r15, 24(%rax) // load lock
```

Metadata Store

Inputs: %rdi - address_of_ptr   %r12 - base

%r13 - bound   %r14 - key

%r15 - lock

```
        movq    %rbx, %rdi
        movq    %rcx, %rbx
        shrq    %rcx, $22
        movabsq %rax, $4398046511096
        andq    %rax, %rcx
        movq    %rcx, _trieroot(%rip)
        movq    %rax, (%rcx,%rax)
        testq   %rax, %rax
        jne     .LBB20_2
# BB#1:
        movq    %rbx, %rbp
        shrq    $25, %rbp
        callq   _trie_allocate
        movq    _trieroot(%rip), %rcx
        movq    %rax, (%rcx,%rbp,8)
        testq   %rax, %rax
        je       abort
.LBB20_2:
        andq    $33554424, %rbx
        movq    (%rax,%rbx,4), %r12
        movq    8(%rax,%rbx,4), %r13
        movq    16(%rax,%rbx,4), %r14
        movq    24(%rax,%rbx,4), %r15
```

Figure 6.2: The x86 instructions inserted by the compiler instrumentation for metadata loads and stores. Metadata store initializes the trie entry if it is NULL.

The metadata accesses inserted before pointer loads/stores introduce instruction overhead and extra control flow. First, there are extra instructions to perform translation from an address of the pointer to its disjoint metadata space address. This involves some arithmetic shift operations, loading the entry in the first level trie, and few more bit-mask and shift operations to obtain the metadata space address as described in Chapter 4. Occasionally it may also involve setting up the first level trie entries if they are absent. Second, there are four loads/stores to access the base, bound, key and lock metadata from the metadata space after the translation. Figure 6.2 illustrates the x86 instructions in our compiler prototype to perform the metadata accesses. Metadata stores have additional overhead to set up the first level trie with a call to *trie_allocate* function as shown in Figure 6.2. Four extra loads/stores with each pointer load/store can exert significant pressure on the cache ports.

Accessing the shadow stack as part of pointer metadata propagation is another source of overhead. Shadow stack accesses are introduced both at the call-site and the callee to store and load the metadata for the pointer arguments respectively. These accesses are optimized away when the function is inlined. However, functions that are not inlined and recursive functions require accesses to the shadow stack.

Beyond the obvious extra instructions inserted by the compiler instrumentation, there are many second-order instruction overheads that result as a side-effect. The extra metadata for the pointers in registers introduces more register pressure because there are more live variables in contrast to the original program. On an x86-64 machine with just 16 general purpose registers, an increase in register pressure can introduce extra spills and restores adding to the instruction overhead. Further, such spills put pressure on the cache ports increasing the performance overheads. Another second order effect is the cache pollution due to extra metadata, which shares the cache with the program's data potentially increasing the number of cache misses.

Among these sources of overheads, our analysis shows that the checks and the metadata loads/-stores are common in contrast to the shadow stack accesses and the second-order effects. The next section describes the acceleration for the checks and the metadata loads/stores. *Watchdog* described in Chapter 5 explored an alternative design point that avoided the shadow stack and some of the second order effects of spilling and restoring by performing instruction injection within the hardware on mostly unmodified binaries.

6.2 New Instructions for Accelerating Checking and Metadata Lookups

This section explores the ISA extensions that are sufficient to reduce the overhead of a state-of-the-art compiler instrumentation described in Chapter 4 to be fast enough to be used in production. We focus on mitigating two sources of overhead: (1) the spatial and temporal checks performed on memory accesses and (2) the metadata lookups that involve a translation from the monitored program memory address to a shadow memory address that maintains information about the monitored program along with loads/stores. To address these overheads, we propose the following new instructions: (1) a spatial check instruction that accelerates the bounds check performed conceptually before every memory access, (2) a temporal safety check instruction that accelerates the load and compare operations performed for enforcing temporal safety, (3) a metadata load instruction to load

64-bits of metadata from the disjoint metadata space, and (4) a metadata store instruction to store 64-bits of metadata to the disjoint metadata space.

The new instructions replace several x86 instructions with a single instruction, thereby enabling the compiler to generate streamline code. Moreover, without the custom check instructions, the extra instructions from the checks increase the number of live variables increasing the register pressure. The metadata load and store instructions provide two benefits: (1) they eliminate the bitmask and shift operations used in the translation from the program address to the metadata address, and (2) they enable us to move from trie data structure to a linear array of bytes (eliminates the first level trie load and the trie root load) as the disjoint metadata space similar to the hardware implementation in Chapter 5. An alternative to these metadata access instructions would be either (1) the instructions that are used to access the shadow space organized as a linear array of bytes but do not perform the translation or (2) the instructions that accelerate the translation overhead with the metadata accesses while still using a two level trie data structure. We do not explore these alternatives. The proposed new metadata load and store instructions transfer the burden of maintaining the metadata space from the compiler/runtime to the hardware. As a result, the hardware can place the metadata space using the unused upper order bits of the virtual address space. Using these metadata access instructions, translation overhead is reduced. Even with these new metadata access instructions, there are still four individual metadata loads/stores to access the metadata which can exert pressure on the cache ports. We investigate the use of vector extensions available in current processors to eliminate multiple loads/stores using wide loads/stores using XMM registers in Section 6.4. Next, we describe the interface and implementation details of the new instructions added to accelerate the *SoftBoundCETS* instrumentation described in Chapter 4.

6.2.1 *SChk* Instruction

To accelerate bounds checks, we add a new instruction, *SChk*, to the x86-64 ISA. This instruction replaces five x86 instructions used earlier (cmp, br, lea, cmp, br) in the bounds check with a single instruction. We describe the ISA interface to the compiler and the implementation details below.

The bounds check performed for spatial safety has four inputs: the pointer being checked, the base, the bound, and the size of the access. The x86 ISA allows us to encode only two register inputs along with an immediate field. Hence, our new *SChk* uses two input registers, immediate and an implicit register (*%rax*) to specify the inputs. Using an implicit register can cause the code generator to emit register-register moves. Unlike the normal instructions, Intel's latest SIMD extensions namely the Advanced Vector Extensions (AVX) allow three input registers. Here, we want to restrict ourselves to the integer pipeline without using the SIMD extensions. We explore the use of SIMD extensions in Section 6.4.

To summarize, the *SChk* instruction takes the pointer in the *%rax* register, the base and the bound in any 64-bit register and the size of the memory access provided as an immediate. The instruction does not produce any output. When the check fails, the instruction causes an exception. In an alternative implementation, the instruction can set the condition code or act as a taken branch on bounds violations.

Implementation *SChk* does not produce any value that is used by the program and is not on the critical path. However, to avoid clogging up the processor bandwidth and resources, *SChk* cracks to a single micro-operation (*uop*) and performs three register reads internally to read the inputs. The extra logic added compares the base with the pointer and the bound with the pointer with the added offset in parallel. As the size of the access can only be 1/2/4/8/16, a simple increment adder is sufficient. As this instruction does not produce a value, the latency of the instruction can be slightly higher, and is not crucial in obtaining low overheads. Figure 6.3 summarizes the instruction interface and provides the details of the implementation.

Comparison to the x86 Bounds Instruction *SChk* is similar in spirit as the `bound` instruction available on x86 processors since the 80286, but *SChk* is different in two key ways. First, *SChk* uses registers to hold all inputs. In contrast, early x86 processors had just 8 registers and no 64-bit datapaths, so the *bound* instruction required both the base and bound to be fetched from memory for each check. As a result, when the base and bound were already in registers, *bound* instruction was typically more expensive than a bounds check using other x86 instructions. The presence of wider datapaths in modern processors allows *SChk* to avoid memory access. Second, *SChk* efficiently supports the byte-granularity checking used by bounds checks. Byte-granularity checking provides the ability to flag a four-byte memory access to a three-byte allocation as an error, but not flag a two-byte access to the same address. *SChk* facilitates this by including the size of the memory access as an immediate. In contrast, the x86 *bound* instruction was designed for checking at the granularity of

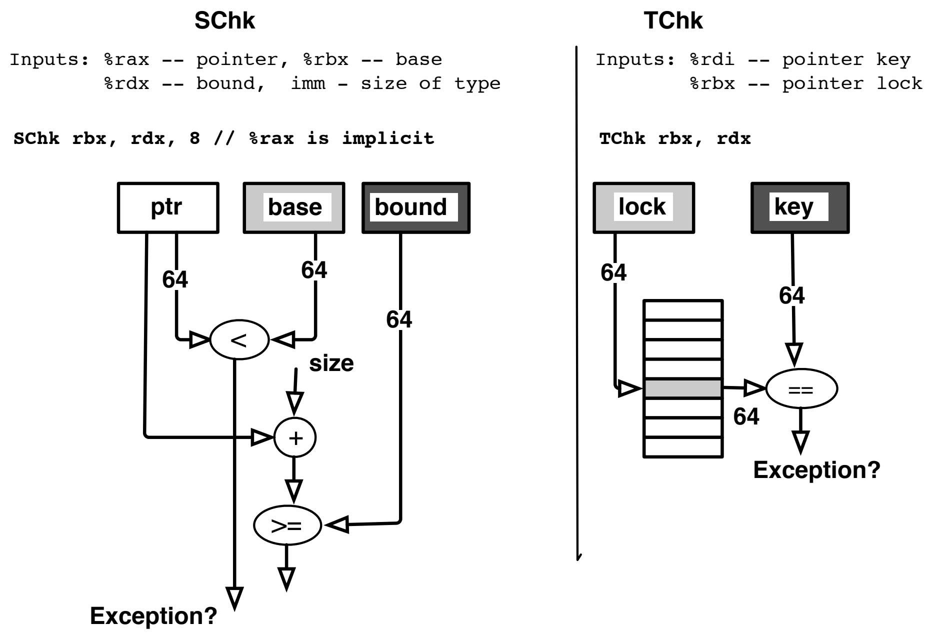


Figure 6.3: Operation of the *SChk* and *TChk*

an array index. Thus, using *bound* in bounds check would require additional instructions to adjust the upper bound based on the size of the memory access.

6.2.2 *TChk* Instruction

To accelerate the temporal pointer checks using the lock and key approach described in Chapter 3, we introduce a new instruction *TChk*. The *TChk* instruction enables us to replace three x86 instructions (load, compare, branch) in Figure 6.1 with a single instruction, reducing the instruction overhead.

The *TChk* takes two register inputs, which contains the key and the lock in any 64-bit register. The instruction does not produce any output. The instruction performs a load using the lock part of the input and checks that the loaded value is equal to the key. When the check fails, it raises an exception.

Implementation The *TChk* instruction does not produce any output that influences the data-flow of the program and is not latency critical. This instruction cracks into a single *uop* that performs a load using the normal load datapath and uses an extra ALU to compare the the loaded value with the register input. This instruction can be cracked into two *uops*, with the first uop performing the load and then the second doing the comparison. Intel's more recent x86 cores would likely fuse them. Hence, fusing these uops will not affect capacity and the bandwidth. Figure 6.3 summarizes the instruction interface and provides the details of the implementation.

6.2.3 Revisiting Metadata Organization for New Metadata Instructions

Beyond the checks, another significant source of instruction overhead is attributed to metadata loads and stores. The instruction sequence generated by the compiler to perform metadata loads/stores is shown in Figure 6.2. There are twelve x86 instructions to perform the metadata load. There are fourteen x86 instructions to perform the metadata store in the common case (when the first level entry is allocated). Many of these instructions perform memory accesses along with the computation. Hence each such instruction internally cracks into multiple micro-operations exerting further pressure on the hardware resources.

There are two major sources of this overhead: mapping the program address to the metadata address and four loads/stores subsequently to load/store the metadata. Mapping also involves many bitmasks and shifts operations to calculate the offset. The mapping also involves two loads: a load to fetch the root of the trie and a load to fetch the second level trie entry. Further, these are dependent chain of loads serializing the processor pipeline.

We make the following observation that directs the design of the new metadata access acceleration instructions. We resorted to a trie lookup structure in our compiler instrumentation described in Chapter 4 because shadowing the entire virtual memory as a linear array of bytes was difficult with current operating system memory management support. If the hardware manages the disjoint metadata space, then we can use linear shadowing for the metadata space. Based on the above observation, we transfer the responsibility of maintaining the disjoint metadata space from the compiler to the hardware. The hardware can use the few upper order bits of the virtual address space to shadow the entire virtual address space. Further, hardware can organize the disjoint metadata space as a linear array of bytes. The operating system (OS) changes are required to make the OS aware of such a disjoint metadata space. The OS changes would allow these shadow accesses to be handled as normal memory accesses using the usual address space translation and page allocation mechanisms. With a shadow space organized as as linear array of bytes, the mapping operation reduces to a few bitmask operations on the pointer address and a subsequent addition of the resulting offset to

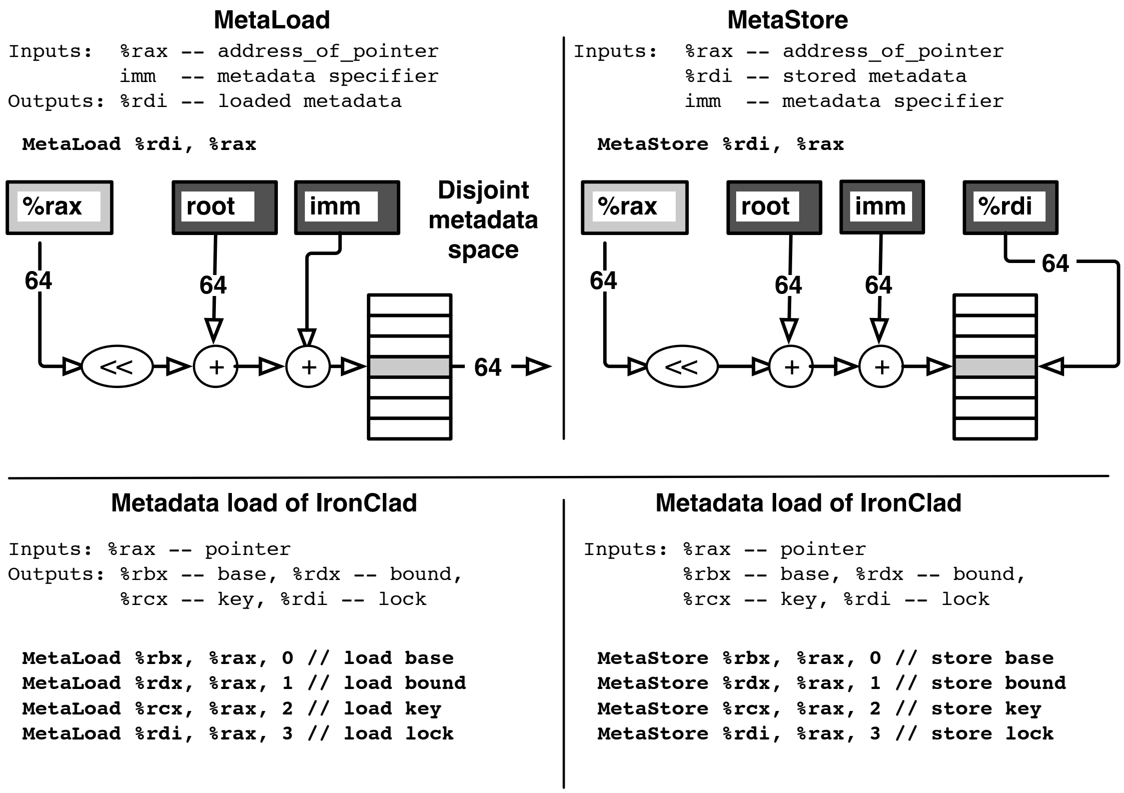


Figure 6.4: Operation of the *MetaLoad* and *MetaStore* instructions. Resultant metadata loads/stores with the instructions is shown.

---

the root of the metadata space. We add a dedicated register to hold the root of the metadata space. We describe the implementation of the metadata acceleration instructions in the subsections below.

6.2.4 *MetaLoad* Instruction

To accelerate the metadata loads performed by the *SoftBoundCETS* instrumentation, we propose a *MetaLoad* instruction that replaces the twelve x86 instructions with four *MetaLoad* instructions as shown in Figure 6.4. We describe the interface to the compiler and the implementation details. The *MetaLoad* instruction takes three inputs: two register inputs—a 64-bit register holding the address of the pointer and a implicit disjoint metadata space root register (*root*)—and an immediate specifying the component of metadata to be loaded. The *MetaLoad* instruction loads the metadata component in the specified 64-bit register.

Implementation This instruction internally cracks into a single *uop* that performs address calculation and a load of the metadata component specified by the immediate field. The hardware calculates the metadata address by multiplying the pointer input address by the size of the metadata to generate an offset that is added to the root register. Finally the address is shifted by a mask depending on the immediate to obtain the resultant metadata component address. Subsequently the hardware issues a request to load the metadata component at the specified address. The bitmask, shift and resultant address generation can be performed in the address generation stage similar to the address generation tasks performed with the normal load instructions in displacement and scaled addressing modes. The *MetaLoad* instruction incurs a latency of four cycles when the load instruction hits in the cache (one cycle for the address generation + three cycles for the load). Figure 6.4 summarizes the instruction interface and provides the details of the implementation. Figure 6.4 also illustrates how the old metadata load in the *SoftBoundCETS* instrumentation becomes four *MetaLoad* instructions instead of twelve x86 instructions in the compiler instrumentation. Although with the new instructions, we still have four instructions for the metadata load, the benefits are substantial because we have eliminated two loads (one for the trie root and the other for the first level trie entry) and six other mask and shift operations as illustrated in Figure 6.2. Eliminating the dependent load (metadata load depends on the first level trie entry) likely translates into performance benefits.

6.2.5 *MetaStore* Instruction

To accelerate the metadata stores performed by the *SoftBoundCETS* instrumentation, we propose a *MetaStore* instruction that replaces the fourteen x86 (in the common case) instructions with four *MetaStore* instructions as shown in Figure 6.4. We describe the interface to the compiler and the implementation details of *MetaStore* instruction. The *MetaStore* instruction takes four register inputs—a 64-bit register holding the address of the pointer, a implicit disjoint metadata space root register (*root*), a 64-bit register holding a component of the metadata and an immediate specifying the metadata component—and produces no output.

Implementation This instruction internally cracks into a single *uop* that performs address calculation and a 64-bit store. The address is calculated by multiplying the pointer input address by the size of the metadata to generate an offset that is added to the root register to obtain the metadata address. The immediate specifies which component of the metadata to be stored. Subsequently the hardware issues a request to store the metadata at the specified address. Figure 6.4 summarizes the instruction interface and provides the details of the implementation. Figure 6.4 also illustrates how the old metadata store in the *SoftBoundCETS* instrumentation can be replaced by four *MetaStore* instructions.

In this section on hardware acceleration for the compiler instrumentation, the compiler instrumentation maintained the components of the metadata in individual registers. As a result, to load all pieces of metadata there were four *MetaLoad* instructions. Similarly, there were four *MetaStore* instructions to store the metadata. Next, we explore the opportunities for reducing such extra metadata loads and stores by using the SIMD 128-bit XMM registers and the related new instructions to accelerate memory safety checking.

6.3 Packed Metadata to Reduce Register Pressure and Multiple Loads/Stores

SoftBoundCETS transformation associates word-sized base, bound, key and lock metadata with each pointer. These extra pieces of metadata cause an increase in the number of live variables, which could potentially result in spills and restores during code generation. To reduce register pressure, this metadata can be be maintained and propagated in packed format throughout the program both in registers and in memory. These packed values are created on pointer creation and unpacked only when the individual values are required during the dereference checks.

SIMD Extensions and XMM Registers on Modern Processors Most processor vendors have added SIMD (single instruction, multiple data) extensions to improve the throughput of floating-point operations in domains such as digital signal processing and graphics. SIMD instructions can greatly increase performance when the exact same operation is performed on multiple data objects. Intel added Streaming SIMD Extensions (SSE) to its x86 ISA starting with Pentium III series of processors to accelerate floating-point operations. Subsequently, SSE was expanded in its revisions (SSE2, SSE3, and SSE4) with more instructions and registers. SSE extensions in the 64-bit operating mode provides sixteen additional 128-bit registers known as XMM0-XMM15. These new registers with the SSE extensions increase the number of logical registers available in the x86-ISA from sixteen to thirty two (16 XMM registers + 16 64-bit registers: RAX, RBX, RCX,

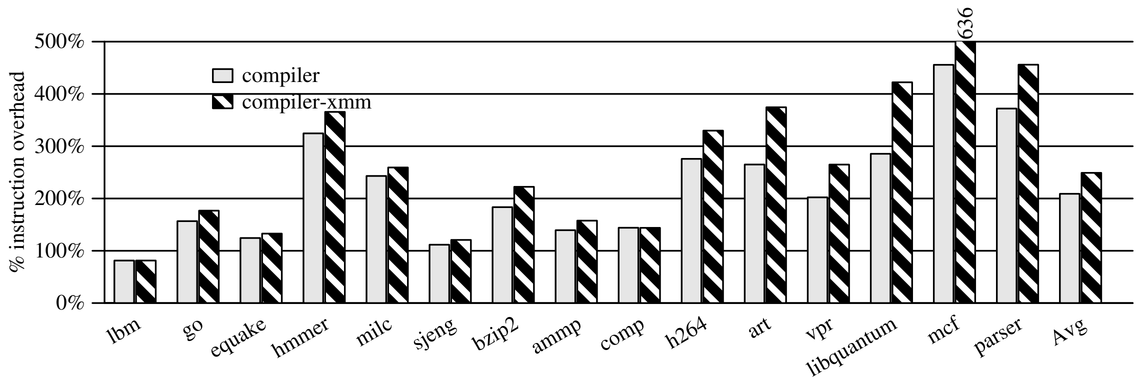


Figure 6.5: Instruction overhead with *SoftBoundCETS* instrumentation (compiler) and *SoftBoundCETS* instrumentation with packed metadata(compiler-xmm).

RDX, RBP, RSP, RDI, RSI, R08-R15). These 128-bit registers are saved and restored across context switches by the operating system like normal registers. We propose the use of metadata in these XMM registers in a packed format in this section to accelerate pointer-based checking. The latest AVX extension by Intel adds sixteen 256-bit YMM registers. Although, we don't use the YMM registers in this section (primarily because the extensions are new and our modified tool chain — binutils — does not support it), the use of YMM registers can further minimize the register pressure and reduce the number of metadata loads and stores compared to the use of XMM registers.

Extending *SoftBoundCETS* to use XMM Registers We modify the compiler instrumentation described in Chapter 4 to use a packed value for the metadata associated with the pointers. In the compiler instrumentation, we create temporaries that have an aggregate type with 128-bits. As we use x86 in our experiments, we keep the packed values resident in the 128-bit wide XMM registers in Intel's x86-64 architecture when the pointer is in a register and use a wide XMM load/store for metadata operations. Using packed values for the metadata reduces the number of load and store operations for the metadata by half when pointers are loaded/stored from/to memory. Apart from the advantages described above, there is an initial penalty for packing the metadata into an aggregate type and later extracting them on pointer dereferences. Each such operation results in *movd* and *punpckhqdq* with *movd* instructions to extract the lower and higher 64-bits respectively on an Intel's x86-64 machine.

Figure 6.5 presents the instruction overhead for the compiler instrumentation that uses normal registers and the 128-bit XMM registers for the metadata. There are two bars for each benchmark. The left bar and the right bar present the instruction overhead of the compiler instrumentation with metadata in normal registers and XMM registers respectively. On average, the compiler instrumentation's instruction overhead is 208% on average. The average instruction overhead of XMM mode compiler instrumentation is 249%, which is slightly higher than the instrumentation with normal registers. Primary source of this instruction overhead is the unpacking operations performed on spatial and temporal checks. Next, we propose custom instructions to perform such checking without unpacking to reduce instruction overhead both due to checks and pack/unpack operations.

6.4 New Instructions for Packed Metadata Checking and Metadata Loads/Stores

This section describes the new instructions for accelerating checks and metadata loads/stores when the compiler instrumentation uses packed metadata. We reorganize the metadata space as linear array of bytes managed by the hardware as described earlier in Section 6.2.3. In the following subsections, we describe the instructions to accelerate spatial checks, temporal checks, metadata loads and stores analogous to the ones described in Section 6.2.

6.4.1 *SChkXMM*

To accelerate bounds checks with packed metadata, we add a new instruction, *SChkXMM*, to the x86-64 ISA. The *SChkXMM* instruction takes the pointer in any 64-bit register, the base and bound in a packed format in any 128-bit XMM register and the size of the memory access is provided as an immediate. Similar to the *SChk* instruction, this instruction does not produce any output and raises an exception on bounds violation. The usage of packed XMM registers avoids the need for implicit registers in the *SChk* proposed earlier, which can eliminate the register-register moves generated before the check due to implicit registers.

The *SChkXMM* cracks to a single *uop* like the *SChk* instruction described earlier. This instruction performs two register reads: 1) a normal register read and 2) an XMM register read. The extra logic added compares the lower half of the XMM register and the pointer and in parallel compares

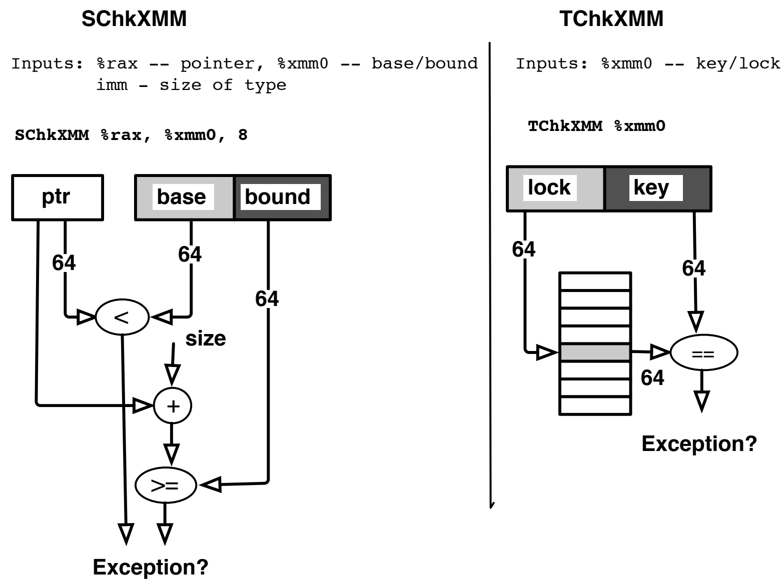


Figure 6.6: Operation of the *SChkXMM* and *TChkXMM* instructions.

the upper half of the XMM register and the pointer with the added offset. If the normal physical register and XMM physical registers are in two register files, the latency of the instruction may be slightly higher. This instruction avoids the unpacking instruction overheads with spatial check when the compiler instrumentation uses packed registers. Figure 6.6 provides the interface and implementation of the *SChkXMM*.

6.4.2 *TChkXMM* Instruction

Similar to the *TChk* described earlier, the *TChkXMM* instruction accelerates the lock and key checking. The *TChk* takes a single input which contains the key and the lock in a packed format in any XMM register. The instruction does not produce any output. Its implementation is similar to the implementation of *TChk* with a single *uop* the performs the loads and compares the key. Figure 6.6 shows the interface and implementation of the *TChkXMM* instruction.

6.4.3 *MetaLoadXMM* and *MetaStoreXMM* Instructions

We propose two new instructions *MetaLoadXMM* and *MetaStoreXMM* to accelerate the metadata loads and stores similar to the *MetaLoad* instruction and *MetaStore* instruction described earlier. The *MetaLoadXMM* instruction takes three inputs: the address of the pointer in any 64-bit regis-

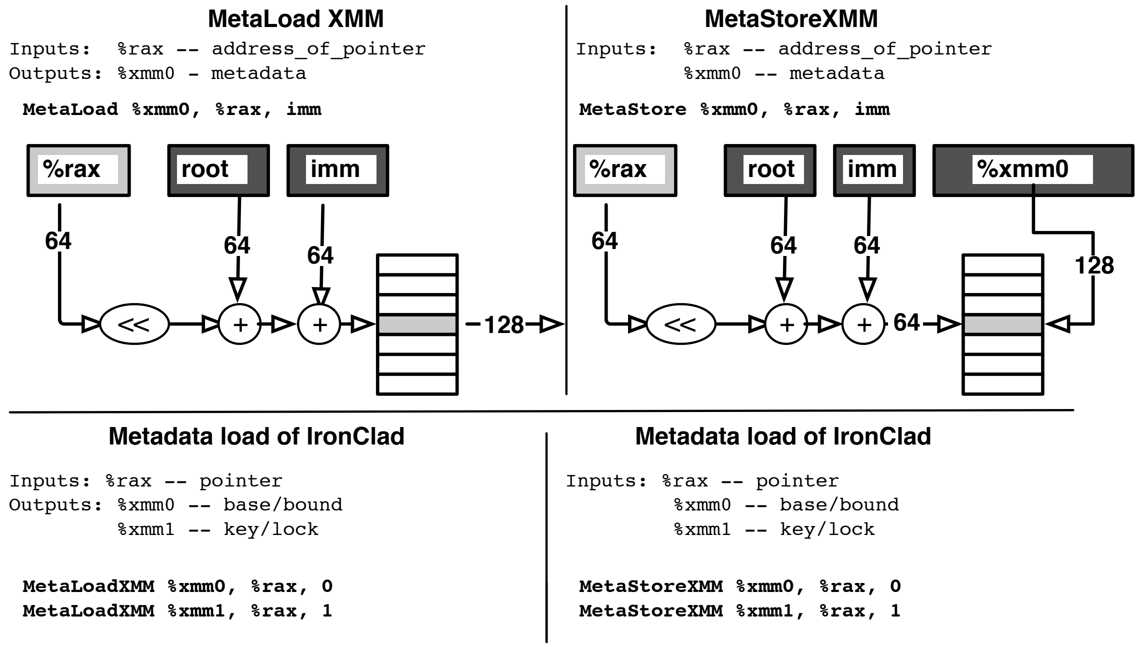


Figure 6.7: Operation of the *MetaLoadXMM* and *MetaStoreXMM* instructions. Resultant metadata loads/stores with the instructions is shown.

ter, a implicit root register holding the beginning address of the metadata space, and a immediate specifying whether the base/bound is being loaded or the key/lock is being loaded. The instruction outputs the loaded metadata in any XMM register. The instruction performs the translation to the disjoint metadata space as described earlier. The *MetaLoadXMM* instruction replaces twelve x86 instructions with the compiler instrumentation with just two instructions. Each *MetaLoadXMM* instruction internally cracks into two micro-operations that perform the individual 64-bit loads. In a streamlined implementation, it can be implemented as a single micro-operation with a wide load.

Similarly, *MetaStoreXMM* instructions performs a wide metadata store to the disjoint metadata space. The instruction takes four inputs: address of the pointer in any 64-bit register, implicit root register to the beginning of the metadata space, wide metadata in any XMM register, and immediate specifying whether the base/bound or key/lock metadata is being stored. This instruction is internally implemented as two micro-operations that performs the translation and the 64-bit stores. Figure 6.7 shows the operation of these instructions. This *MetaStoreXMM* instruction replaces fourteen XMM instructions for the compiler instrumentation to just two instructions.

6.5 Summary of Hardware Changes

One of the goals with *HCH* is to minimize the hardware changes. The new instructions described in this chapter require a few changes to the processor core when compared to the traditional pipeline. *HCH* only requires changes to the instruction decoder to decode the newly added instructions. As hardware vendors regularly add new instructions and there are already more than one thousand instructions in the ISA, the additional complexity to add the proposed instructions is small. Beyond the changes to the decoder, additional control logic is needed to implement the functionality for the added instructions. In summary, *HCH* minimizes the hardware changes by (1) leveraging the compiler instrumentation to perform majority of the work (such as identifying pointers, propagating metadata and metadata creation), (2) keeping the hardware changes localized to the instruction decoder, and (3) proposing nominal additional logic to implement the functionality. Next, we provide a qualitative comparison of the *HCH* instrumentation proposed in this chapter and the *Watchdog* instrumentation for instrumenting unmodified binaries described in Chapter 5.

6.6 Qualitative Comparison of *HCH* and *Watchdog*

We looked at two points in the design space for enforcing memory safety using a pointer-based checking approach with *Watchdog* in Chapter 5 and with *HCH* in this chapter. We highlight the key differences between the two design points and summarize the advantages and disadvantages of these two design points in this section.

6.6.1 Differences between *HCH* and *Watchdog*

The two design points for hardware support—*Watchdog* and *HCH*—differ from each other in four main design choices. First, these design points differ in the approach adopted to add instrumentation for pointer-based checking. *HCH* uses new instructions that are inserted by the compiler's backend to instrument the code. On the other hand, *Watchdog* uses micro-operation (*uop*) injection to perform pointer-based checking transparently on binaries. Second, they differ in the mechanisms used to identify pointers to perform pointer-based checking. *HCH* relies on the pointer information available to the compiler in the intermediate representation as described in Chapter 4. There are two alternatives with *Watchdog*: (1) it can use heuristics to identify pointers using conservative pointer identification or (2) it can use ISA-assisted pointer identification that require tool chain changes. Third, *HCH* and *Watchdog* differ on who manages the pointer metadata propagation for register operations. With *HCH*, the compiler propagates metadata for register operations, eliminates redundant copies by performing copy elimination for free. In contrast, *Watchdog* performs metadata propagation for pointer operations in registers using *select uops* and physical register sharing extensions to the maptable in the processor. Fourth, *HCH* and *Watchdog* differ in how the metadata gets propagated for pointer arguments and returns on function calls. *HCH* relies on the shadow stack mechanisms provided by the compiler instrumentation to propagate metadata likely introducing more loads and stores. On the other hand, *Watchdog* just relies on pointer metadata propagation for register operations to propagate metadata.

Having described the key differences between the design points for hardware support, we recount the advantages and disadvantages of the two design points in the next two subsections.

6.6.2 Advantages and Disadvantages of *HCH*

The advantages of *HCH*-like hardware support are described below. First, a key advantage of *HCH* is that it can leverage precise pointer information from the compiler to perform pointer-based checking. Second, *HCH*-like hardware support can leverage compiler optimizations to perform check elimination and metadata propagation optimizations. Third, *HCH* can leverage any guidance provided by the programmer to classify safe accesses to avoid checks and propagation. Fourth, the hardware support for *HCH* is extremely small with a couple of additions to the decode stage of the pipeline to add new instructions. Fifth, the performance overheads are reasonably competitive.

However, all the above described advantages of *HCH* come with a price of significant changes to the tool chain. The programs need to be recompiled with a memory safety compiler as described in Chapter 4. Beyond tool chain changes, the *HCH* hardware support inherits the inefficiencies of the compiler instrumentation such as the metadata propagation for pointer parameters using a shadow stack and the extra spilling, which is a result of few architecturally visible registers. Further, passing pointer arguments and returning pointers with function calls becomes reasonably expensive with the use of the shadow stack for metadata propagation.

6.6.3 Advantages and Disadvantages of *Watchdog*

The advantages of *Watchdog* are described below. First, *Watchdog* can operate on mostly unmodified binaries avoiding the tool chain changes. This is especially attractive for legacy code where programmers do not want to either change the code or distribute the source code. Second, although pointer identification with binaries is hard, simple heuristics are reasonably effective. Pointer metadata propagation for register operations ensure that calls and returns are not expensive. Fourth, *Watchdog*'s *uop* injection has a large number of physical registers at its disposal and avoids the spilling that would have been generated by a compiler instrumentation. Fifth, *Watchdog*'s overheads are reasonably low with small changes to the hardware even while performing pointer-based checking almost purely in hardware.

Watchdog's primary disadvantage is the inability to leverage either the information available to the compiler or provided by the programmer to identify pointers, eliminate checks at compile time, and classify programs as safe. This disadvantage results in the *Watchdog* hardware performing metadata propagation and checking on every memory access including spills and restores inserted by the compiler.

6.7 Support for Multithreading

Any approach for providing memory safety would ideally detect all errors even when the program is multithreaded. If operations on pointers, its metadata, and checks occur non-atomically, interleaved execution and race conditions (either unintentional races or intentional races used in lock-free concurrent data structures) can result in both missing errors and false violations. To enable our approach to work seamlessly with multithreaded workloads, the implementation should ensure the following requirements:

- **Requirement#1:** A pointer load/store's data and metadata accesses execute atomically.
- **Requirement#2:** The spatial and the temporal check should execute atomically with the load/store operation.
- **Requirement#3:** The runtime needs to allocate metadata in a thread-safe way

- **Requirement#4:** Each thread needs to use thread-local shadow stack for propagating metadata with pointer arguments and return values on function calls.

Enforcement of these requirements may need additional mechanisms described below depending on whether the program is well-synchronized and is free of data races. There are three such cases: (1) when the multithreaded program is well-synchronized and has no races either with data variables or synchronization variables, (2) when the program is well-synchronized, and has no data races but uses atomic operations such as compare-and-swap (CAS), and (3) when the program is not well-synchronized and has data races.

6.7.1 Handling Well-Synchronized Programs without Data Races

When the program is well-synchronized, free of data races and does not use atomic operations such as compare-and-swap (CAS) explicitly in the program, the additional mechanisms required to enforce memory safety are simple. As the program is well-synchronized, every pointer dereference is within a critical section. As long as our instrumentation introduces extra operations within the critical section, the atomicity of the pointer load/store with the metadata is ensured satisfying requirement#1 described above. Similarly, the atomicity of the temporal check and the spatial check is also ensured by the adding extra instrumentation within the critical section satisfying requirement#2.

To satisfy requirement#3, our runtime needs to allocate identifiers in a thread-safe way. The runtime partitions the space of identifiers with each thread using identifiers from a thread-local pool to allocate identifiers in a thread-safe manner. Further, the updates to the first level trie entries with the compiler instrumentation, which are allocated on demand, need to be performed in a thread-safe manner. To ensure that first level trie entries are allocated in a thread-safe manner, we create a mapping with *mmap* for the second level entry and use a compare-and-swap operation to set the first level trie entry. In the common case, when two threads do not conflict on the same first level trie entry, the compare-and-swap operation succeeds. In the case where two threads are attempting to update the same first level trie entry, one thread fails to successfully complete compare-and-swap operation. The failing thread releases the mapping that was allocated by it before the failing compare-and-swap operation and loads the mapping for the second level entry from the first level trie.

To satisfy requirement#4, each thread has a thread-local shadow stack to propagate metadata for pointer arguments and return values with function calls. Since the additional mechanisms to enforce safety are simple, there is likely no additional performance overhead for this class of programs.

6.7.2 Handling Well-Synchronized Programs with Synchronization Races

For the second class of programs that are well-synchronized without any data races but explicitly use atomic operations such as the compare-and-swap, we need additional mechanisms to ensure the atomicity of the metadata load/store and the pointer dereference involved in a compare-swap-operation. All pointers accessed without the atomic primitives (*e.g.* compare-and-swap) are data race free as the program is well-synchronized. For such pointer operations, the implementation can insert additional instrumentation within the critical section as described above. The instrumentation allocates metadata in a thread-safe way and use a thread-local shadow stack as described above. To maintain atomicity of the pointer access in a compare-and-swap operation with the metadata accesses and the checks, the instrumentation can perform the following code transformations: (1) introduce a new critical section, (2) replace the compare-swap-operation with regular loads/stores and a compare operation introduced within the newly inserted critical section, and (3) introduce the metadata space accesses and the spatial and temporal checks within the introduced critical section thereby satisfying the requirement#1 and requirement#2 described above.

To ensure that such transformations do not slow down the program (which could occur if we use a single lock to implement the critical section for this transformation), the implementation can maintain a table of locks and the transformation selects a lock based on the address of the pointer involved in the atomic operation. As all threads acquire the locks introduced by instrumentation in the same order for the compare-and-swap-transformation, they do not introduce any new cyclic dependences and deadlocks. In the presence of hardware support, this transformation can be implemented using a bounded-transaction where the disjoint metadata accesses and the temporal check are performed in a transaction along with the original compare-and-swap instruction. Intel has recently announced best-effort transactional memory support in the next generation of processors code-named Haswell (being released in 2013), so such atomic-updates are implementable.

6.7.3 Handling Programs with Data Races

For the third class of programs that are not well-synchronized and have data races, the C memory model does not provide any semantics and the behavior of such programs is undefined according to the standard. To ensure that the program is free of memory safety errors, an implementation must satisfy the four requirements described above. The implementation can easily satisfy requirement#3 and requirement#4 by allocating metadata in a thread safe manner.

The key challenges are in ensuring the atomicity of the checks with the pointer accesses and the metadata accesses with the pointer accesses. In the absence of such atomicity with the metadata access and the pointer access, racy accesses can manufacture metadata by splicing metadata from the pointer operations involved in a race. For example, if two threads are loading pointer *p* and storing pointer *q* to the same location *addr* simultaneously, the resultant metadata loaded by a thread could have the old *base* metadata stored at the disjoint metadata associated at address *addr* and the new *bound* metadata from the pointer *q* (and similarly all other combinations of metadata splicing) potentially missing memory safety violations.

Among the two checks, the spatial check has its inputs (the base, the bound, and the pointer) in registers and does not perform any memory access. Hence, the spatial check does not need any special mechanism. However, the temporal check can fail to detect some temporal-safety violations as a result of the vulnerable time window between the time of a temporal check and the actual pointer dereference. Other racy accesses can deallocate the memory between the vulnerable time-window.

The atomicity of the metadata accesses and the temporal checks with the pointer dereferences can be ensured in one of the following two ways. First, the instrumentation can transform the program to introduce a new critical section, perform the memory access, metadata accesses and the temporal check within the critical section as described above for handling compare-and-swap operations. However this transformation is likely to introduce a large performance overhead. To reduce the overhead, the instrumentation can be coupled with a static data race detector with new critical sections introduced only for racy accesses. Second when the hardware support is available, a bounded-transaction mechanism (like in Haswell) can provide the required atomicity.

Even in the absence of such bounded-transaction support, the pointer-based checking with disjoint-metadata will not miss any violations if the instrumentation ensures that metadata is loaded/stored as an atomic single 256-bit wide load/store and the temporal check is performed after the pointer access. An atomic load/store to access the metadata ensures that the subsequent optimizations (performed either by the hardware or the compiler) will not reorder the metadata load/store. A single wide load/store ensures that the metadata does not get spliced from the racy accesses. Hence, the racy program will not be able to manufacture arbitrary metadata for the pointers. As program contains data races, racy accesses can still cause pointer and metadata mismatches, resulting in only false violations; not a compromise in system security. Further, the program needs to tolerate a single dangling access as the temporal check is performed after the pointer dereference.

6.8 Experimental Evaluation

This section provides an experimental evaluation of *HCH* highlighting (1) its effectiveness in reducing the instruction overheads and (2) the resultant performance overheads. The next few subsections describe our experimental setup, the benchmarks, changes made to the entire tool chain — compiler, assembler, and the simulator — and the experimental methodology.

6.8.1 Experimental Methodology

We use the *SoftBoundCETS* prototype described in Chapter 4 to insert the new instructions. We also extended the instrumentation to store the metadata in a packed format. We leverage all the optimizations described in Chapter 4 to accelerate the checking. We use the simulator described in Section 5.5 of Chapter 5. The out-of-order core parameters are available in Table 5.1 of Chapter 5. We changed the cracker to decode the new instructions and changed the scheduler to perform the port reservations and to account for the addition of an extra ALU after the load datapath for the added instructions.

Benchmarks We chose the CPU-intensive benchmarks written in C from SPEC benchmarks suite as described in Section 4.4 of Chapter 4 for testing the performance overheads. To evaluate *HCH*, we made significant changes to the tool chain—compiler, binutils, and simulator—, and our infrastructure is not yet robust enough to handle all the benchmarks. Hence, we report the results for the benchmarks currently working in our prototype. We used the same inputs for all the benchmarks as with our *Watchdog* performance evaluation.

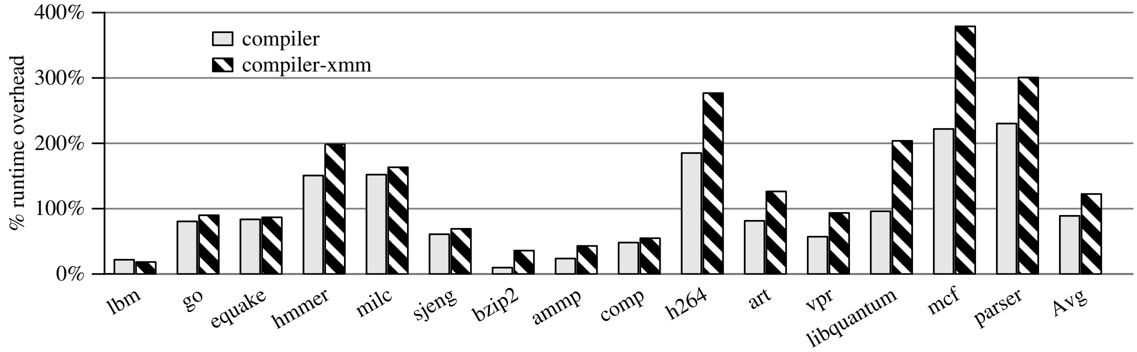


Figure 6.8: Execution time overhead of the compiler instrumentation (compiler) without and with packed metadata in a XMM register (compiler-xmm).

Sampling We adopt a different sampling methodology compared to *Watchdog* evaluation as the binary being simulated would vary with and without addition of new instructions to propagate and check pointer metadata. We ran the benchmarks with *randomized* 2% sampling with each sample of 10 million instructions preceeded by a fast-forward and a warmup of 480 and 10 million instructions respectively. As we are inserting extra code to perform the checks, the binaries created are different. To enable a fair comparison, we ran each benchmark multiple times, with each run having a significant number of randomized samples. The mean of the execution time (calculated from the macro-operation IPC and the total number of instructions executed) from multiple runs are used in the data reported in this paper. Our experiments show that the standard deviation in the execution time across the multiple randomized runs (10 runs) is small and is on average less than 1%.

6.8.2 Runtime Performance Overhead with Packed Metadata

Figure 6.8 reports the execution time overhead of the compiler instrumentation (*SoftBoundCETS* described in Chapter 4) both without and with packed metadata using the x86 simulator and the methodology reported in Section 6.8.1. The overheads reported in Figure 6.8 are slightly different from the compiler instrumentation overheads in Chapter 4 because of the following two reasons. First, the experiments use different inputs resulting in slightly different overheads. The experiments reported in this chapter use the train inputs to perform simulations to keep the simulation time reasonable. Second, the complete details of the SandyBridge Core i7 chip are not publicly available.

We model it based on the publicly available information. Small deviations in our simulator model and the actual chip can account for the difference in behavior.

The average performance overhead for the compiler instrumentation using the simulator is 89%. The average performance overhead for the compiler instrumentation with packed metadata is higher at 122% as a consequence of packing and unpacking operations performed during the spatial and temporal checks. As the compiler instrumentation in scalar mode has lower performance overhead, we use it as a reference to compare the performance improvements with the addition of new instructions in the rest of the evaluation unless explicitly stated otherwise.

6.8.3 Overheads with New Instructions

Figure 6.9 reports the execution time overhead of the compiler instrumentation with new instructions. There are three bars for each benchmark. The height of the leftmost bar is the execution time performance overhead of the compiler instrumentation. The height of the middle and rightmost bars present the execution time overhead of the compiler instrumentation with all the new instructions in scalar mode and in packed mode with XMM registers respectively. The new instructions in scalar mode reduce the performance overhead to 45% on average (from 89% without instructions). The packed mode new instructions using XMM registers reduce the performance overhead to 39% on average. The use of custom new instructions in packed mode gets substantial benefit by not only accelerating the checks but also removing the unpacking operations and reducing the register pressure. Further, packed mode instructions also eliminate the register-register moves resulting due to the use of implicit register with *schk* instruction. Benchmarks *milc* and *libquantum* attain significant speedups with the new instructions. On the other hand, *mcf* and *parser* do not experience large speedups with new instructions as they have a large number of function calls resulting in shadow stack accesses. Further, metadata with pointers exerts pressure on the cache resulting in extra overhead.

The performance improvements with the new instructions are correlated with the reduction in the number of instructions executed. Figure 6.10 presents the instruction overhead for the compiler instrumentation with and without the new instructions. There are three bars for each benchmark. The height of the leftmost, middle and rightmost bar reports the instruction overhead in scalar mode, with all the new instructions in scalar mode and with all the new instructions in packed mode

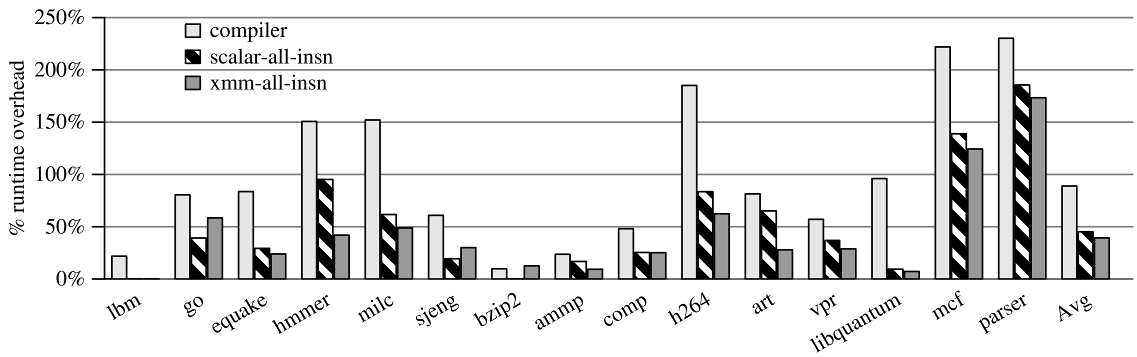


Figure 6.9: Execution time performance overhead of the compiler instrumentation (compiler) with all the instructions in scalar mode (scalar-all-insn) and with all the instructions in packed mode (xmm-all-insn).

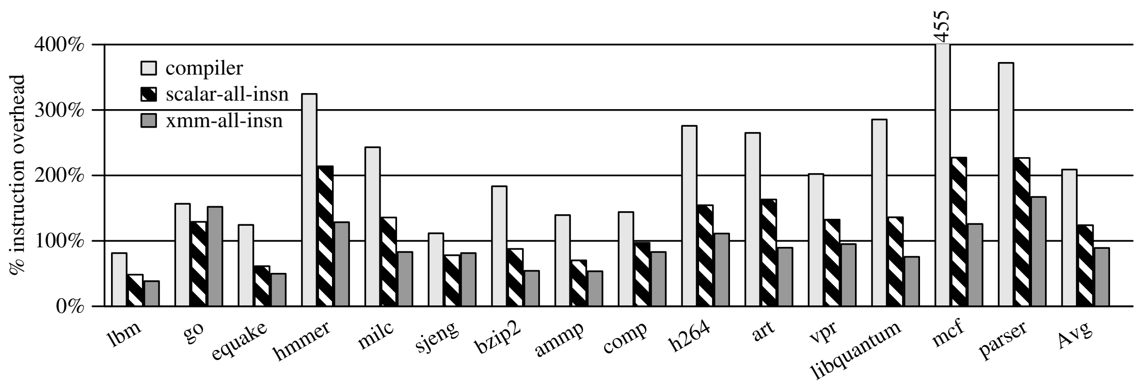


Figure 6.10: Instruction overheads of the compiler instrumentation in scalar mode (compiler), with all the new instructions in scalar mode (scalar-all-insn) and new instructions in packed mode (xmm-all-insn).

respectively. The average instruction overhead with all the new instructions in scalar mode reduces to 124% on average (from 208% instruction overhead with compiler instrumentation). The average instruction overhead with all the new instructions in packed mode reduces to 89% on average. The reduction in the number of instructions and the number of live variables (hence the spills and restores) enables packed mode with all the new instructions to attain better performance overheads compared to the new instructions in scalar mode.

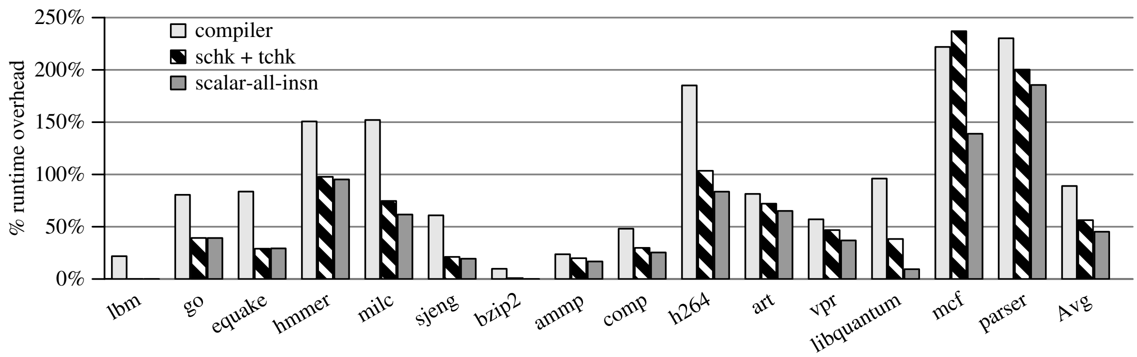


Figure 6.11: Execution time performance overhead of the compiler instrumentation (1) without new instructions (compiler), (2) with just the check instructions (schk + tchk), and (3) with all the instructions in scalar mode (scalar-all-insn).

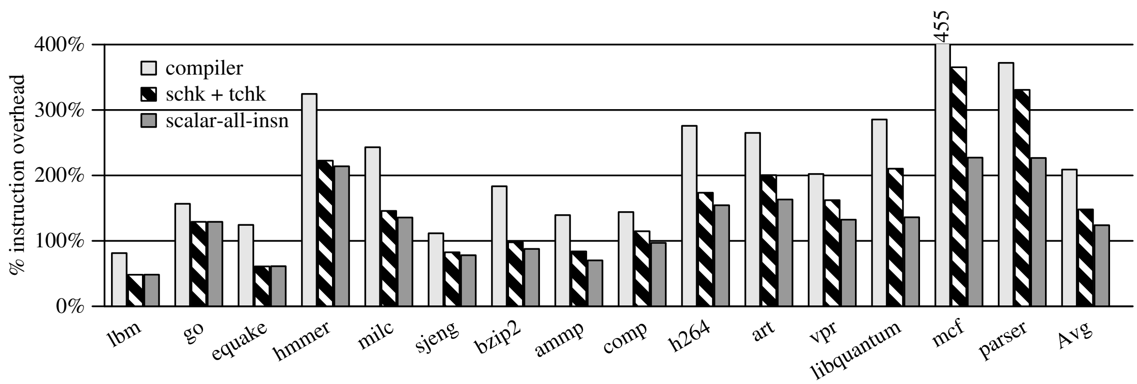


Figure 6.12: Instruction overheads of the compiler instrumentation in scalar mode (1) without new instructions (compiler), (2) with the new check instructions (schk + tchk), and (3) all the instructions (scalar-all-insn).

6.8.4 Impact of Various Instructions

This subsection presents the impact of individual check and metadata access acceleration instructions on the performance overhead and the instruction overhead.

Checks vs Metadata Accesses in Scalar Mode Figure 6.11 reports the execution time overhead of the check instructions and the metadata access instructions in scalar mode. There are three bars for each benchmark with the height of the left-most bar, the middle bar and the right-most bar representing the performance overhead of the scalar compiler instrumentation without any new instructions, with just check instructions(*schk* + *tchk*) and all the instructions (checks + metadata access) respectively. The average performance overhead with just the check instructions reduces to 56% (from 89% with no instructions). For the benchmarks on the left of the Figure 6.11 that execute few pointer loads and stores, the check instructions provide significant benefit. The difference in the height of the middle and right-most bars provides the benefits of adding the metadata access instructions. On average, addition of the metadata access instructions reduced the overhead from 56% (with just check instructions) to 45%. The benchmarks on the right of the Figure 6.11 that perform slightly higher number of metadata accesses obtain benefits from the metadata access acceleration instructions.

Figure 6.12 reports the instruction overhead for the compiler instrumentation without instructions (leftmost bar), with check instructions (middle bar) and all the check and metadata access instructions (rightmost bar). Addition of new check instructions reduce the average instruction overhead to 148% (from 208% with no instructions). Adding the metadata access instructions along with the check instructions reduces the instruction overhead to 123% on average. Benchmarks that have large reductions in instruction overhead report large performance improvements in Figure 6.11.

Spatial Check vs Temporal Check in Scalar Mode Figure 6.13 reports the execution time performance improvements due to the addition of individual check instructions. There are four bars for each benchmark. The height of the leftmost and right-most bars represent the performance overhead of the compiler instrumentation in scalar mode without any instructions and with both the check instructions respectively. The height of the left middle bar and the right middle bar represent the execution time performance overhead of just the spatial check instruction and the temporal check instruction respectively. We find that among the two check instructions, spatial check instruction (schk) reduces the performance overheads significantly. On average, the spatial check instruction reduces the performance overhead from 89% to 60%. On the other hand, temporal check alone reduces the performance overhead from 89% to 87%.

Figure 6.14 reports the instruction overheads for the individual check instructions in scalar mode. On average, spatial check instruction reduces the instruction overhead from 208% to 158%. On average, the temporal check instruction reduces the overhead from 208% to 201%.

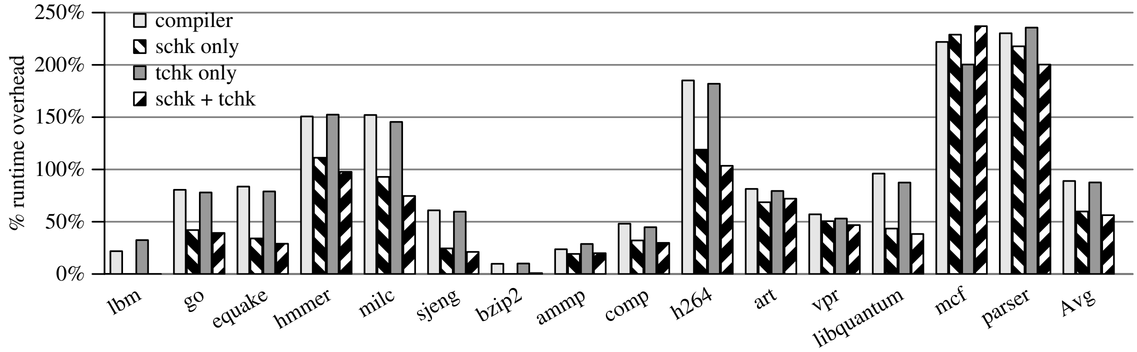


Figure 6.13: Execution time performance overhead of the compiler instrumentation (1) without new instructions (compiler), (2) with just the spatial check instruction (schk only), (3) with just temporal check instruction (tchk only), and (4) with both the check instructions in scalar mode (schk + tchk).

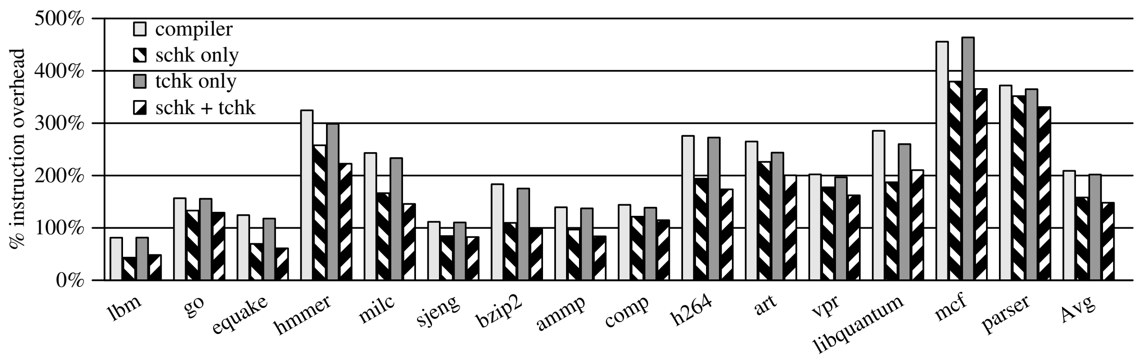


Figure 6.14: Instruction overheads of the compiler instrumentation in scalar mode (1) without new instructions (compiler), (2) with only the spatial check instruction (schk only), (3) with only the temporal check instruction (tchk only), and (4) both the check instructions (schk + tchk).

Checks vs Metadata Accesses in Packed Mode Figure 6.15 reports the performance overhead of the compiler instruction (1) without new instructions in packed mode (leftmost bar - compiler-xmm), (2) with just check instructions in packed mode (middle bar - schx + tchx), and (3) with all the new instructions in packed mode (rightmost bar- all-insn). Addition of check instructions in packed mode reduces the overhead to 62% on average (from 122% with packed mode compiler instrumentation). Elimination of numerous instructions necessary to unpack the metadata along with the acceleration for the checks is the contribution for this reduction. Addition of metadata instructions along with the check instructions reduced the overhead to 39% on average. The new metadata access instructions in packed mode reduce the performance overhead better than the new

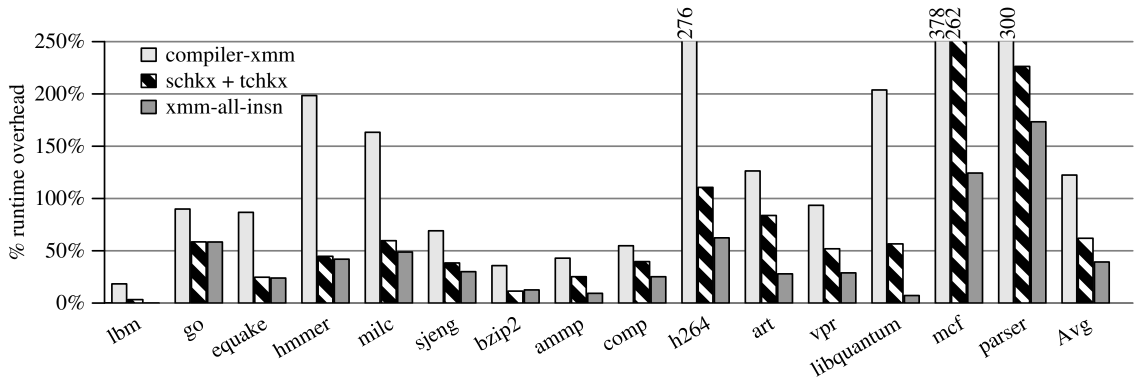


Figure 6.15: Execution time performance overhead of the compiler instrumentation in packed mode (1) without instructions (compiler-xmm) with just the check instructions (schx + tchkx) and with all the instructions in packed mode (xmm-all-insn).

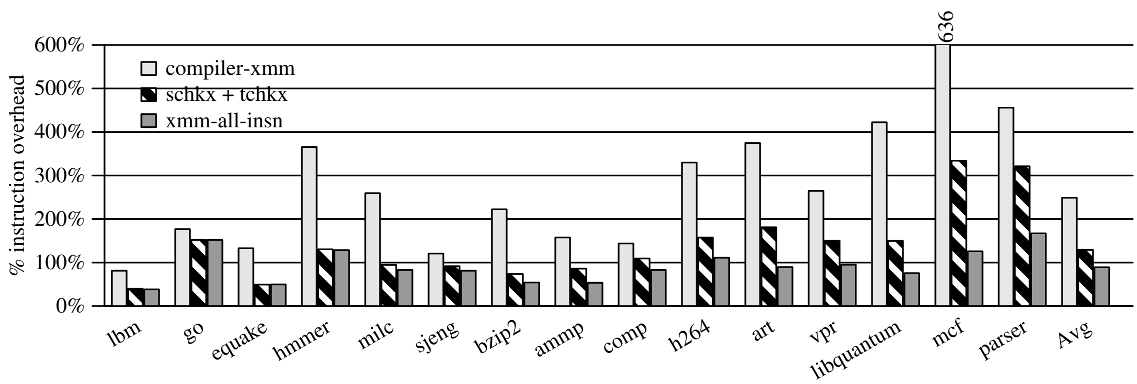


Figure 6.16: Instruction overheads of the compiler instrumentation in packed mode (1) without new instructions (compiler-xmm), (2) with the new check instructions (schkx + tchkx) and all the new instructions (xmm-all-insn).

metadata access instructions in the scalar mode. This is primarily attributed to the reduction in the register pressure and the resultant spills. Although internally each metadata access instruction in packed mode cracks into two micro-operations, reducing the number of live variables reduces the register pressure and the second order effects.

Figure 6.16 presents the instruction overhead for the compiler instrumentation in packed mode (1) without instructions (leftmost bar - comp-xmm), (2) with check instructions (middle bar - schx + tchkx), and (3) with all the new instructions (rightmost bar). On average, the check instructions reduce the instruction overhead from 250% (with no instructions) to 129%. Addition

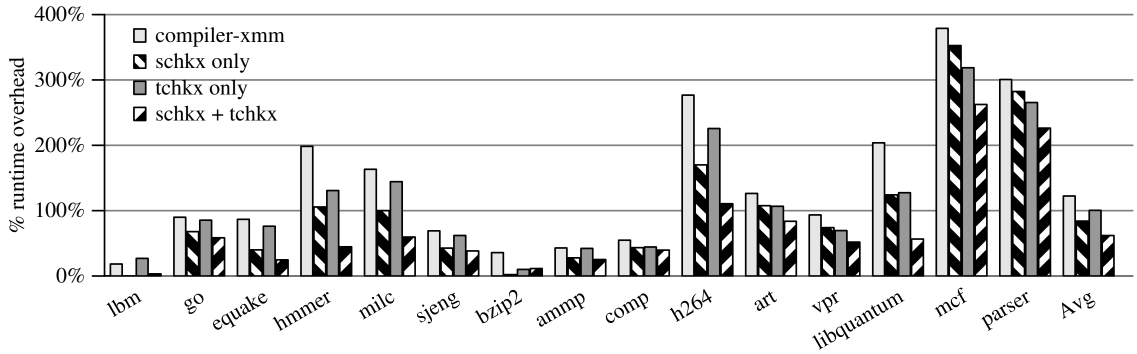


Figure 6.17: Execution time performance overhead of the compiler instrumentation (1) without new instructions (compiler-xmm), (2) with just the spatial check instruction (schkx only), (3) with just temporal check instruction (tchkx only), and (4) with both the check instructions in packed mode (schkx + tchkx).

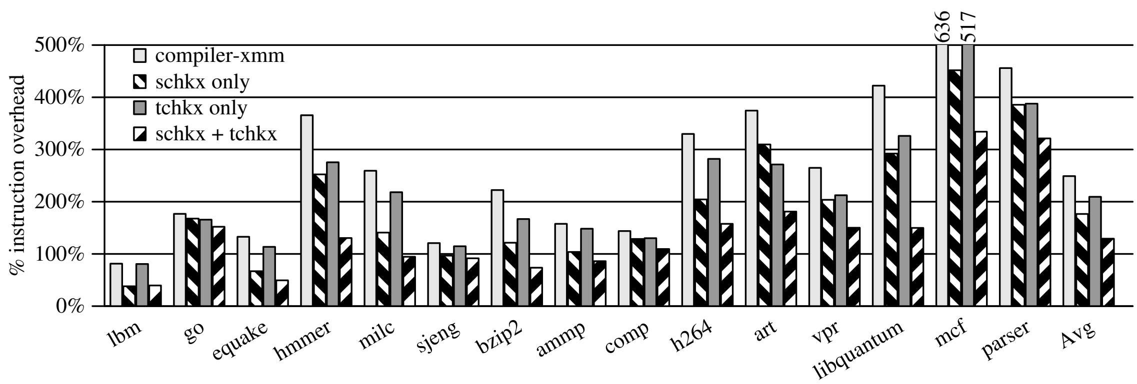


Figure 6.18: Instruction overheads of the compiler instrumentation in packed mode (1) without new instructions (compiler-xmm), (2) with only the spatial check instruction (schkx only), (3) with only the temporal check instruction (tchkx only), and (4) both the check instructions (schkx + tchkx).

of metadata access instructions along with the check instructions reduces the instruction overhead further from 129% (with check instructions) to 89%.

Spatial Check vs Temporal Check in Packed Mode Figure 6.17 reports the execution time performance improvements due to the addition of individual check instructions in packed mode. There are four bars for each benchmark. The height of the leftmost and right-most bars represent the performance overhead of the compiler instrumentation in packed mode without any instructions and with both the check instructions respectively. The height of the left middle bar and the right middle

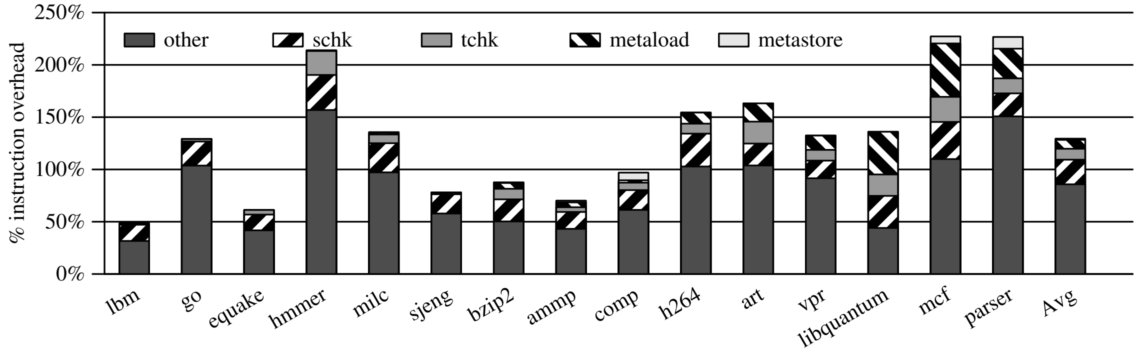


Figure 6.19: Breakdown of the remaining source of instruction overhead in scalar mode.

bar represent the execution time performance overhead of just the spatial check instruction and the temporal check instruction respectively. Similar to our findings with the check instructions in scalar mode, we find that spatial check instruction (schkx) provides more performance benefit than the temporal check instruction (tchkx). On average, the spatial check alone reduces the performance overhead from 122% to 84%. On the other hand, temporal check alone reduces the performance overhead from 122% to 100%. In packed mode, temporal check provides more performance benefits than scalar mode as it avoids the unpacking operations before the check.

Figure 6.18 reports the instruction overheads for the individual check instructions in packed mode. On average, spatial check reduces the instruction overhead from 250% to 176%. On average, the temporal check instruction reduces the overhead from 250% to 209%. Elimination of unpacking operations in packed mode compiler instrumentation with the new check instructions provides more instruction overhead benefits compared to the scalar mode.

Remaining Source of Instruction Overhead in Scalar Mode Figure 6.19 reports the remaining sources of instruction overhead with the new instructions in the scalar mode. The height of each bar represents the total instruction overhead with all the new instructions in scalar mode. Each bar has five segments. The length of bottommost segment represents the instruction overhead resulting as a consequence of pointer-based checking with disjoint metadata apart from the instruction overhead due to the checks and the metadata loads/stores. The contribution from the other extra instructions is the major source of remaining instruction overhead for most benchmarks. There are two main reasons for this high other extra instruction overhead. First, the *HCH* instrumentation relies on the

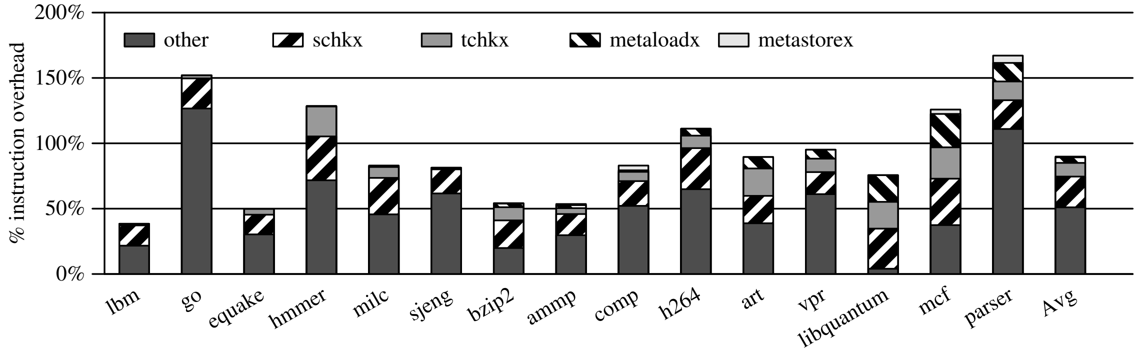


Figure 6.20: Breakdown of the remaining source of instruction overhead in packed mode.

---

compiler to propagate the metadata for pointer arguments and return values using the shadow stack, which introduces instruction overhead. Second, the increased register pressure with the increase in the number of live variables with metadata temporaries result in stack spills and restores contributing to the instruction overhead. On average, the instruction overhead from other extra instructions alone is 80%.

The second and the third segment from the bottom represent the contribution of the new spatial check (*schk*) and the temporal check (*tchk*) instruction to the instruction overhead in the scalar mode. The top two segments in each bar represent the contribution to the instruction overhead from the new metadata load and store instructions in the scalar mode. Among the instructions added to accelerate pointer-based checking, spatial check instruction is the primary contributor to the remaining instruction overhead. The benchmarks on the right that have a large number of metadata loads/stores incur a significant fraction of the instruction overhead from the metadata load/store instructions. On average, *schk*, *tchk*, *metaload*, and *metastore* instructions contribute 24%, 11%, 8% and 1% to an aggregate instruction overhead of 124% in scalar mode.

Remaining Source of Instruction Overhead in Packed Mode Figure 6.20 reports the remaining sources of instruction overhead with new instructions in the packed mode. The height of each bar represents the total instruction overhead with all the new instruction in packed mode. Each bar has five segments. The length of bottommost segment represents the instruction overhead apart from the checks and metadata loads/stores. Like in the scalar mode in Figure 6.19, the other extra instructions due to the shadow stack accesses and the spills are significant. However, packing the metadata in

XMM registers has reduced the amount of instructions. On average, the other extra instructions result in 51% instruction overhead. Packing the metadata in 256-bit wide registers (YMM registers) will further reduce the instruction overhead from the extra instructions.

The second and the third segment from the bottom represent the contribution to the instruction overhead from the new spatial check (*schkx*) and the temporal check (*tchkx*) instruction in the packed mode. Like in scalar mode in Figure 6.19, the *schkx* and *tchkx* contribute 24% and 11% respectively on average to the instruction overhead.

The top two segments in each bar represent the contribution to the instruction overhead from the new metadata load and store instructions in the packed mode. The benchmarks on the right that have a large number of metadata loads/stores see a significant reduction in their contribution to the instruction overhead as the number of metadata loads/stores are halved with packed loads/stores. On average, *metaloadx* and *metastorex* contribute 4% and 0.5% to an aggregate instruction overhead of 89% in packed mode. Further the instruction overhead due to these metadata load/store instructions can be halved by packing the metadata in 256-bit wide registers.

6.9 Summary

This chapter presented hardware acceleration in the form of new ISA extensions for the compiler instrumentation proposed in Chapter 4 for performing pointer-based checking with disjoint metadata. We observed that a large number of instructions are executed to perform checks and metadata accesses. To alleviate the resulting instruction overheads, we proposed new instructions to perform spatial checks, temporal checks and the metadata accesses. We also transferred the burden of managing the metadata space from the compiler to the hardware that enabled the use of flat linear array as the disjoint metadata space. Using the new instructions, we reduced the instruction overheads and resulting performance overheads of pointer-based checking approach significantly. We also found that using the SIMD extensions on modern machines by packing the metadata in SIMD registers is attractive when complemented with new check and metadata access instructions.

Chapter 7

Related Work

The problem of memory safety with C is an old and a well researched topic. We described the closely related approaches for enforcing memory safety in Chapter 2. In this section, we describe other efforts in either retrofitting memory safety or in defending against the security vulnerabilities that result as a consequence of memory safety errors. In this chapter, we describe the language extensions and static analysis based solutions (Section 7.1), probabilistic approaches using randomization (Section 7.2), dynamic checking solutions (Section 7.3), hardware-based solutions (Section 7.4), and general acceleration to mitigate bottlenecks (Section 7.5) to provide varying degrees of memory safety enforcement either in a direct or an indirect manner.

7.1 Language Extensions and Static Analysis Based Solutions

There have been numerous efforts to design dialects of C, which can be easily checked statically or with dynamic checking to enforce either partial or complete memory safety [35, 53, 63, 73, 94, 122].

Polymorphic C [122] presents a provably type-safe dialect of C that includes most features of C except casts and structures. However, a well-typed Polymorphic C program can still fail to terminate and can have memory safety errors due to dangling pointer errors.

Cyclone [73] is a new dialect that attempts to be closely compatible to C. Cyclone introduces annotations with pointer type declarations, replaces C-style unions with ML-style sum types, and performs pointer-based checking to provide safety. Cyclone uses garbage collection to avoid temporal safety errors. Subsequent research from the Cyclone team has attempted to alleviate the problems with garbage collection using region-based memory management [61].

CCured [94] discussed in detail in Chapter 2 is a backwards-compatible extension to C that use whole type inference to infer pointer kinds and performs dynamic checking to retrofit spatial safety. CCured relies on the use of a garbage collector to prevent temporal safety errors. Deputy [35], a follow-up work from the CCured team, is a dependently typed system that uses programmer annotations to bound pointers and tag unions, while avoiding whole program type inference and reducing the runtime overhead of enforcing spatial safety. Like CCured, Deputy does nothing for providing temporal safety and hence, needs to be coupled with a garbage collector.

Evans [53] describes a set of programmer annotations to make program assumptions explicit at interface points such as function calls. Using these annotations, the program can be checked locally to detect violations of programmer assumptions such as null pointer dereferences and some dangling pointer dereferences.

To enable modular checking for C, SAL [63], an annotation language widely used at Microsoft allows programmers to specify requirements for safely using each buffer in the program using lightweight annotations. Further, a significant fraction of these annotations have been inferred using various program analysis tools. The key emphasis with this project has been on incremental deployment of annotation-based checking to detect future violations of memory safety in production code at development.

In contrast to programmer provided annotations, Dhurjati *et al.* [47], have created a subset of C language that provides partial memory safety without runtime checks or garbage collection. To provide spatial safety, they rely on type safety, disallow casts and restrict the array indices to be simple affine expressions. To provide temporal safety, they rely on automatic pool allocation [84] that assigns all heap-allocated objects to type homogeneous pools, where each member has the same type and alignment. On reallocation of freed memory, new object will have the same type and alignment as the old object. The key idea in this line of work is to prevent violations of type-safety through dangling pointer errors rather than detecting dangling pointer errors. To prevent temporal errors on the stack, they use data structure analysis [82] that performs a flow-insensitive and context sensitive analysis to identify all pointers to the stack frame that escape into heap or global locations.

To avoid annotations, many static analyses that detect buffer overflows without programmer annotations have been proposed. These include tools using abstract interpretation [22, 48] and integer programming [56] techniques tailored to a specific domain. Static analysis has also been coupled with lightweight programmer or inferred annotations as described above [35, 63]. Static checking tools generally either have false positives [129] or false negatives (they are incomplete), but are certainly useful complementary techniques to dynamically enforced spatial memory safety. Further, many of the language extensions can be used with the pointer-based checking proposed in this dissertation to further reduce overheads.

7.2 Probabilistic Approaches with Randomization

Probabilistic approaches randomize the allocation of objects making memory safety errors hard to exploit. They either use an infinite heap or approximate it with various kinds of randomization. However, masking the errors rather than detecting them can make debugging difficult.

The three common types of randomization are address-space randomization, instruction-set randomization, and data-space randomization. Address-space-layout randomization (ASLR) randomizes the location of the objects on the stack, heap and the libraries in the address space of the program. The key advantage of ASLR is that it is simple to use and cheap. Though reasonably effective, attackers have succeeded in exploiting the memory safety errors by spraying the objects on the heap. Instruction-set randomization (ISR) [19, 77] randomizes the presentation of code. ISR creates process-specific randomized instruction set of the system executing potentially vulnerable software. Randomizing an arbitrary instruction set (the x86 machine code) involves three components: the randomizing element, the execution environment (*e.g.*, an appropriately modified processor), and the loader. An attacker who does not know the key to the randomization algorithm will inject code that is invalid for that randomized processor, causing a runtime exception.

In contrast to ASLR and ISR, data-space randomization (DSR) [21, 25] randomizes the representation of different objects stored in memory. These schemes modify the data representation xoring each data object in memory with a unique random mask (encryption), and to unmask it before its use (decryption). DSR can be implemented using a program transformation that modifies each assignment $x = v$ in the program into $x = mx \oplus v$, where mx is a mask associated with the variable x. DSR defeats buffer overflow attacks by preventing the write of the value intended by the attacker. For example in a buffer overflow attack that overwrites an adjacent variable b with a value v, the value written to that location will be $v \oplus ma$ with DSR. On a subsequent read, the value read will be $(v \oplus ma) \oplus mb$, which would be a random value rather than the attacker intended value. DSR provides more entropy than ASLR and ISR, and is able to thwart more attacks.

Other Probabilistic approaches [20, 79, 99, 101] use the idea of an infinite heap to detect memory safety errors. By allocating two objects infinitely apart and not reusing memory, memory safety violations cannot corrupt allocated objects. Practical implementations approximate the infinite heap abstraction either by allocating objects randomly or by randomizing the reuse of memory. These approaches are attractive as they can be applied to existing systems easily without significant tool chain changes. However, as these schemes mask errors rather than exposing them, debugging memory safety errors can become difficult.

The open-source tools Electric Fence [1], PageHeap [5], and DUMA [17] allocate each object on a different physical and virtual page. Upon deallocation, the access permission on individual virtual pages are disabled, increasing memory usage and causing high runtime overheads for allocation-intensive programs [45]. A spatial safety violation or a temporal safety violation will cause a page fault that is appropriately handled. To detect errors in the presence of reallocations, virtual pages cannot be reused. Further, as these schemes allocate one object per virtual and physical page, large memory overheads can result for programs with many small objects. More recently, proposals have addressed the physical memory overhead issue by placing multiple objects per physical page, but mapping a different virtual page for each object to the shared page [45, 88].

DieHard [20] provides probabilistic memory safety by approximating an infinite-sized heap using the runtime. It uses a randomized memory manager which places objects randomly across the heap. Randomized allocation makes it unlikely that a newly freed object will soon be overwritten by a subsequent allocation thus avoiding dangling pointer errors. Exterminator [101] builds on top of DieHard, and it carries out error detection and correction based on data accumulated from multiple executions without programmer intervention. Archipelago [88] reduces the memory overhead of DieHard by trading address space for security.

7.3 Dynamic Checking Solutions

A large number of proposals attempt to provide partial or full memory safety by performing varying degrees of checking at runtime. We described the closely related dynamic checking tools in Chapter 2. In this section, we describe some of the other related proposals that have enforced memory safety indirectly by enforcing other program properties [11, 13, 80], and dynamic checking tailored for checking specific kinds of errors.

7.3.1 Pointer Checking Tools

Beyond the common and related pointer-based checking approaches described in Chapter 2, this section describes some of the other efforts (both early and recent) in using pointer-based checking to retrofit safety. The early solutions that performed dynamic checking to provide some safety were primarily debugging tools and had significant overhead. Most of these tools were source-to-source translation schemes and primarily focused on detecting spatial safety violations.

BCC [78] is a source- to-source translator that added calls to a runtime library to perform checks. It changed each pointer to include upper and lower bounds. The BCC reported slowdowns in the range of 30×. Rtcc [123] is similar to BCC but focuses only on adding run-time checking for array subscripts in the Portable C Compiler (PCC). Unlike BCC and Rtcc, SafeC [16] and MSCC [131] described in Chapter 2 could detect both spatial and temporal errors. Samurai [104] attempts to retrofit memory safety for only certain sections of the program by using programmer provided annotations. The programmers annotate certain memory regions as critical for program execution. The runtime ensures that non-critical memory regions do not influence the critical regions using replication and error correction.

Fail-Safe C [102] is a spatial safety compiler for the C language that is fully compatible with the ANSI C specification. The key contribution is the maintenance of metadata with integers. It uses fat pointers, as well as fat integers to allow casts between pointers and integers.

A recent proposal MemSafe [120], uses disjoint metadata with pointers inspired by our spatial safety work on SoftBound [92]. Unlike our approach that maintains different metadata to detect both spatial and temporal errors (Chapter 3), Memsafe uses spatial metadata to detect temporal safety violations. Memsafe sets the bounds metadata to be invalid when the memory is deallocated. To handle aliases, it designs a SSA representation that can be used to invalidate the bounds metadata of all aliasing pointers.

7.3.2 Object-Based Checking Tools

The object-based checking approaches were described in detail in Chapter 2. There has been a steady reduction in the slowdowns reported by these approaches since the original solution by Jones and Kelly [75]. The approach by Jones and Kelly [75] provided backwards compatibility using a splay tree to map memory to its metadata. The checking overheads were high in the range of 10× to 30×. CRED [113] extended Jones and Kelly's approach to handle out-of-bound addresses to compute in-bounds addresses. It achieves this using an auxiliary data structure to store information about out-of-bounds pointers. It also improves on performance at the cost of protection by limiting checking to string operations, thereby reducing the overheads to 20%-130% range. Dhurjati *et al.* [44] partition objects into pools at compile time and uses a splay tree for each pool. These splay trees can be looked up more efficiently than the single splay tree used by previous approaches, and each pool has a cache for even faster lookups reducing the overhead of Jones and Kelly significantly. SAFECode [46] designs an analysis that builds on the automatic pool allocation [84] to maintain the soundness of the compile-time points-to graph, the call graph, and the available type information at runtime even in the presence of memory safety errors. The sound analysis information when coupled with their object-based approach, can reduce the overhead of enforcing memory safety.

Baggy bounds checking [14] uses an object-based approach but enforces allocation bounds instead of object bounds. The key idea in baggy bounds checking is to reduce the overhead of checks with object-based approach by (1) padding and aligning objects to powers of two, and (2) enforcing allocation bounds instead of object bounds with a check that involves a single memory access, a shift and a xor operation. By eliminating splay tree lookups, baggy bounds checking reduces the overhead of object-based approaches significantly. PAriCheck [134] is similar to baggy bounds checking in enforcing allocation bounds instead of object bounds but handles the out-of-bound pointers more efficiently. However, as structure accesses are not checked and only pointer arithmetic is checked (as in object-based schemes), these schemes fails to detect all spatial safety violations.

7.3.3 Enforcing Memory Safety on Binaries

Several tools and approaches have adopted checking on binaries to avoid tool chain changes and recompilation. Purify [65] and Valgrind [97] are two popular tools that are widely used to perform checking on binaries to detect some subset of the memory safety errors. As dynamic binary instrumentation itself adds overheads, these schemes have overheads higher than $10\times$. Further partial checking can be adopted to check vulnerable library functions (strcpy) in preloaded shared libraries. The libsafe [18] tool uses the frame pointer to limit copy operations within the stack frame so as to avoid overwrites of the return address. BodyArmor [121], a recent proposal on detecting buffer overflows in binaries without recompilation reduces the overhead for array access protection to around $2\times$. BodyArmor uses binary reverse engineering tools to identify arrays in the binary, rewrites the binary to assign colors to the arrays and checks the color of the accesses before every memory access. Tools that perform checking on binaries are attractive as they can be easily used out of the box. However, limited protection, and the presence of false positives and false negatives along with high runtime overhead restricts their use.

7.3.4 Indirect Approaches for Memory Safety

Many approaches indirectly provide partial memory safety by enforcing some other program property. Control-Flow Integrity (CFI) [11] is based on the idea that software execution must follow a path of a control-flow graph (CFG) determined ahead of time. The control-flow graph can be obtained by source analysis, binary analysis or execution profiling. CFI is efficiently implemented using binary rewriting. Although useful in thwarting many attacks, many attacks have exploited memory safety errors without violating control flow integrity [31]. Data-Flow Integrity [27] is similar to CFI but instead of computing a CFG ahead of time, it computes a data-flow graph at compile time and instruments the program to ensure that flow of data at runtime follows the data-flow graph. To perform such checking, it combines static points-to-analysis with runtime instrumentation. DFI maintains a table that maps each instruction to its last writer to the location. The table is updated before every write and checked before every read to ensure that last writer is in conformance with the static data flow graph. DFI can detect many both spatial and temporal errors, but the false positives and false negatives depend on the precision of the underlying static points-to-analysis.

Similar to DFI, Write Integrity Testing (WIT) [13] checks only writes and streamlines the implementation by assigning colors to the objects and checks writes. WIT unlike CFI, operates on the source code, and it attains low overheads. WIT also generalizes the work of Yong *et al.* [133] that performs store-only checking to detect security vulnerabilities.

7.3.5 Software Fault Isolation

When a complicated piece of software (*e.g.*, the web browser, the kernel and others) needs to be protected from memory safety errors in its extensions and modules, isolating the errors in the extensions is attractive. Many such software fault isolation systems have been proposed. Software-based fault isolation techniques like SFI [128] and XFI [52] isolate the kernel extensions with a low overhead. SafeDrive [139] uses programmer provided annotations in the extensions to bounds check C code. Although it has low overhead, it provides weak isolation guarantees. Further, these isolation schemes do not detect temporal errors. Similarly, SVA [40] provides a safe execution environment for an entire operating system, and can enforce some safety properties to prevent many memory errors.

In contrast to safety with kernel extensions, Native Client [132] provides a sandboxing facility to run a subset of untrusted x86 code natively within the browser. The goal of Native Client is to provide browser based applications the computational performance of native code. Native Client uses software fault isolation along with a secure runtime to manage interactions and side effects.

7.4 Hardware Support for Memory Safety

Apart from the hardware-based checking schemes described in Chapter 2, we describe few of the other related proposals that enforce memory safety by performing varying degrees of checking in hardware.

7.4.1 Checking with Hardware

HardBound [43] is a hardware-based bounds checking approach previously proposed by our research group. HardBound provided spatial safety by injecting micro-operations for each load and store instruction to perform checks and metadata propagation. HardBound also used the shadow space based disjoint-metadata for maintaining metadata with pointers. As binaries do not contain pointer information, HardBound was forced to perform metadata loads and stores on every memory access. To avoid such overheads, it used a metadata cache that maintained a bit indicating whether the PC accessed a pointer or not. HardBound did not provide protection against temporal safety violations. Unlike HardBound, which focused on in-order processors, *Watchdog* proposed in this dissertation focuses on full memory safety, proposes pointer identification schemes, and proposes register renaming-based techniques to provide low overhead safety on modern out-of-order processors.

The work of Chuang *et al.* [32] is related to our hardware instrumentation *Watchdog*. The work of Chuang *et al.* explores different checking-metadata organizations (per-object or per-pointer), but places at least some metadata with each pointer and object. Like *Watchdog*, they also describe architectural support, including new micro-ops and metadata mapping/caching structures. These structures require maptable-like checkpointing/recovery on branch predictions, and they are architecturally visible (and their state must be saved to memory on every context switch). However, they fail to consider the source-incompatibility implications and loss of comprehensive protection with such in-line metadata in the presence of memory reuse and unsafe type-casts. *Watchdog* differs from Chuang *et al.*'s work in several significant ways. First, *Watchdog* places all metadata in disjoint-metadata spaces, to (1) retain more compatibility with existing programs by leaving the program layout completely unchanged and (2) provide more comprehensive protection in programs with unsafe type-casts from corrupting the metadata. Second, *Watchdog* introduces only simple structures (a data cache for the lock locations, solely to reduce the contention for data cache ports); even without this cache, the overhead increased only from 24% to 29%.

AccMon [140] is a hardware proposal that exploits the property that in most programs, a given memory location is typically accessed by only a few instructions. Therefore, by capturing the invariant set of program addresses that normally access a given variable, one can detect accesses by outlier instructions, which are often caused by memory errors. Although, it can detect common memory errors, it suffers from false positives.

SafeMem [110] is a hardware proposal that provides Purify-like [65] protection but makes novel use of existing ECC memory technology to insert guards around objects with reasonable overhead. SafeMem uses invalid ECC codes to detect accesses via dangling pointers (unless the memory has been re-allocated). Further, SafeMem pads heap allocations with a cache block with invalid ECC to detect contiguous buffer overflows. Like many other proposals, SafeMem also suffers from false positives.

7.4.2 Hardware Acceleration for Garbage Collection

As temporal safety violations can be avoided by using a garbage collector, many hardware acceleration proposals have attempted to accelerate various aspects of garbage collection. Schmidt and Nielsen [114] accelerate garbage collection for real-time applications using garbage-collected memory modules (GCMM), which include both RAM and a microprocessor capable of running a real time copying collection algorithm locally on the GCMM. Although such hardware is attractive, the whole heap needs to be allocated to custom memory modules. Further, it prevents the software from implementing different allocation and collection policies. As an alternative design point, Click *et al.* [34] have proposed Pauseless GC, a concurrent GC with partial compaction supported by efficient hardware read barriers.

Other proposals have explored hardware-based reference-counting to accelerate reference counting based garbage collection. Peng and Sohi [105] use a in-cache reference-counting mechanism to accelerate GC. Subsequent efforts in this direction have used reference counted caches managed by coprocessors that handle object allocation and in-cache collection [28]. Rather than using custom coprocessors, HAMM [74] proposes a flexible acceleration for reference counting based garbage collection that works with a general-purpose architecture using standard memory modules and does not tie the hardware to a particular allocator and GC implementation.

Meyer *et al.* [89] propose a hardware co-processor to perform concurrent garbage collection. They propose a native pointer type in the hardware that can be leveraged to accelerate garbage collection. The use of native pointer type and explicit pointer registers are similar to the ISA-assisted pointer identification proposed with *Watchdog* to accelerate pointer-based checking with disjoint metadata. Existence of pointer registers and instructions that manipulate pointers can avoid the inefficiencies of pointer identification with *Watchdog*.

7.4.3 Taint checking and Intrusion Detection

Instead of enforcing memory safety directly, some proposals attempt to detect malicious code when it is injected into the system, typically by marking some untrusted data as "tainted" and propagating that information through computations on the processor. Some projects in this vein are Minos [39], LIFT [109], RIFLE [125], and Raksha [42]. Other techniques detect anomalous behavior [80, 142] or combine tainting and bounds checking [33] to detect memory safety errors. In contrast to approaches that provide complete memory safety, the taint checking approach may permit a program to overwrite buffers, so long as the data is not provided by some untrusted source. Thus, although information-flow tracking and intrusion detection can stop some forms of malicious code or data injection, they do not prevent all bounds violations that can corrupt data. These approaches do have the complementary advantage in that they are capable of preventing SQL injection, format-string injection, and related attacks in which untrusted inputs cause security violations without breaking memory safety. Further, taint tracking mechanisms are ineffective with temporal safety violations in the presence of memory reallocations.

7.4.4 Cryptographic and Tamper Resistant Hardware

There has also been much recent work on hardware support for cryptographically sealed code [50], encrypted memory [86, 118], secure processors [85, 119, 124], and tamper resistant hardware [86]. Although this work is largely orthogonal to the memory safety support proposed in this dissertation, these techniques do provide tamper resistance and some protection against code injection attacks—the attacker would have to provide code appropriately signed or encrypted in order to inject it into the instruction stream. These techniques are also not intended to protect against all spatial safety and temporal safety errors.

7.5 Other Acceleration Techniques

Apart from performing checking to providing various degrees of memory safety, others have proposed alternative memory management techniques to avoid some temporal safety violations, and general acceleration schemes for various checkers.

7.5.1 Region-Based Memory Management

Region-based schemes [58, 61] allocate the objects in specific regions which can be specified either implicitly or explicitly. When the region is freed, all the allocated objects in the region are freed. To ensure that there are no dangling pointers, these systems maintain information about the set of external pointers pointing to a region by maintaining reference counts and signal an error when the region being freed has a non-zero reference count. Further, HeapSafe [59] uses reference counts and ideas from region management to make manual management safe, but requires modifications to the program source code.

Another approach based on automatic pool allocation [47] assigns all heap-allocated objects to type homogeneous pools, where each member has the same type and alignment. On reallocation of freed memory, new object will have the same type and alignment as the old object. In similar spirit, Cling [12] memory allocator also prevents type unsafe address space reuse among objects of different types. Thus, dereferencing dangling pointers to reallocated memory cannot cause violations of type safety. This approach does not prevent erroneous dereferences through dangling pointers but ensures only that such dereferences do not violate type safety.

7.5.2 Acceleration for Metadata and Checks

A few other researchers [96, 138] have observed mapping shadow memory as one of the performance problems in designing binary translation-based checkers and have proposed efficient software mechanisms to shadow memory on 64-bit machines with reasonable translation overhead [138]. In the context of log-based architecture (LBA), instructions to accelerate translation from program address to metadata address used a separate metadata translation buffer to cache the first level entry of the trie, requiring extra mechanisms to keep it coherent. To identify redundant checks, dynamic filtering [64] adds a new instruction to the ISA, *dyfl*, which identifies such checks using a lossy set based implementation. Though proposed in the context of garbage collection to detect redundant checks in the creation of inter-generational references, it can be applied to other tools and safety related checking. Similarly, idempotent filters and mapping instructions have been proposed in the LBA context [30]. These acceleration schemes are orthogonal to the techniques proposed in this dissertation and can be used to further reduce the overhead of pointer-based checking proposed in this dissertation.

Chapter 8

Conclusions

In this chapter, we first summarize the dissertation in Section 8.1. We briefly describe the details on the recent commercial adoption of the ideas proposed in this dissertation in Section 8.2. We provide directions for future work in the area of enforcing memory safety in Section 8.3. Finally, I describe the lessons I learned in this process and my reflections on the area of memory safety.

8.1 Dissertation Summary

In this dissertation, we addressed the problem of retrofitting memory safety for C programs with the three goals: (1) comprehensiveness, (2) compatibility, and (3) competitive performance. To address this problem, we adopted the pointer-based checking approach that maintains metadata with pointers. We proposed the use of bounds metadata and identifier metadata to provide both spatial and temporal safety. One of the key contributions of this dissertation is to maintain the metadata with pointers in a disjoint metadata space to provide both spatial and temporal safety. The use of disjoint metadata enabled us to revisit pointer-based checking (generally considered invasive) to enforce memory safety. We developed reasoning techniques to illustrate how pointer-based checking with disjoint metadata can provide comprehensive and compatible spatial and temporal safety even in the presence of arbitrary type casts and memory reallocations in the program.

The use of disjoint metadata required us to address three problems: (1) design an efficient organization of the disjoint metadata space, (2) design mechanisms to access the metadata space, and (3) mitigate the performance overheads in accessing the disjoint metadata space. We organized the disjoint metadata space as a table lookup structure (a trie and a shadow space organized as a linear array of bytes). To index into this table lookup structure and access the metadata, our approach uses the address of the pointer rather than what the pointer points to. To mitigate the performance overheads with this disjoint metadata accesses, our approach performs the accesses to the metadata space only when pointers are loaded or stored. However, maintaining metadata only with pointers also results in our approach disallowing the creation of pointers from integers. To provide efficient implementation of this approach with low overheads, we proposed efficient instrumentation purely within the compiler, within hardware and with hybrid hardware compiler support. We also detected both previously unknown (new) and known memory safety errors in applications, benchmark suites, and test harnesses. We have experimented with more than one million lines of existing C code, demonstrating the compatibility of the approach.

Unlike prior pointer-based checking schemes that have been source-to-source translations [94], we proposed *SoftBoundCETS* instrumentation within the compiler to attain low performance overheads. *SoftBoundCETS* leverages the type information in the LLVM intermediate representation to identify pointers and perform the instrumentation on optimized code. Further, *SoftBoundCETS* uses a shadow stack to propagate metadata for pointer arguments and pointer return values with function calls. The shadow stack provides dynamic typing for pointer arguments between the call-site and the callee. A combination of simple local transformation, instrumentation on optimized code and streamlined implementation enabled *SoftBoundCETS* to attain low overheads. *SoftBoundCETS*'s full checking mode that propagates metadata on all pointer operations and performs spatial and temporal checks on loads and stores has a performance overhead of 108% on average with the SPEC benchmark suite. These runtimes overheads are likely more than acceptable for debugging, internal and external testing, and for mission-critical applications. Further, *SoftBoundCETS*'s store-only checking mode that detects security vulnerabilities reduces the average overhead to 41%. The low overheads with store-only checking while detecting all memory safety based security vulnerabilities makes it attractive for active deployment in production code.

To perform pointer-based checking on unmodified binaries with low overheads, we proposed *Watchdog* hardware that injected micro-operations to propagate metadata and perform checks. One of the challenges with *Watchdog* was related to the identification of pointers as binaries lack such information. We designed heuristics and tool chain extensions for identifying pointers and showed that it is effective in identifying pointers and eliminating unnecessary metadata accesses. *Watchdog* streamlines the implementation by eliminating unnecessary dependences between original program micro-operations and the injected micro-operations by maintaining the data and metadata in separate physical registers. *Watchdog* eliminates unnecessary copies of the metadata by extending the register renaming logic to facilitate physical register sharing. Further, we mitigated the structural hazards that result with the use of memory access ports by the temporal safety checks using a lock location cache. *Watchdog* demonstrates that it is possible to perform pointer-based checking purely in hardware on unmodified binaries with 24% runtime overhead (17% for a single check *uop* configuration) for enforcing both spatial and temporal memory safety without sophisticated compiler instrumentation and analyses. *Watchdog* enforces memory safety on unmodified binaries with low overheads making it attractive for deployment in production systems.

Leveraging the insights gained from building *SoftBoundCETS* instrumentation and *Watchdog* instrumentation, we proposed *HCH* instrumentation to minimize the hardware changes compared to *Watchdog* while providing significant acceleration for *SoftBoundCETS*. We observed that most of the hardware investment in *Watchdog* was used to identify and propagate metadata with pointers. To reduce the hardware changes, *HCH* leverages the compiler instrumentation that already has pointer information to identify and propagate metadata with pointers. The hardware accelerates the checks and the metadata accesses with new instructions, which are used by the compiler instrumentation. Further, the new instructions can be implemented with localized changes to the instruction decoder. We showed that *HCH* instrumentation can leverage the vector registers on modern machines to pack the metadata and reduce the register pressure. *HCH* instrumentation with new instructions reduces the overhead of pointer-based checking to 39% on average. These overheads will reduce further with the new vector extensions that provide 256-bit registers. In summary, *HCH* provides full memory safety with reasonably low overheads while incurring low hardware costs and using a simple local instrumentation, which makes *HCH* attractive for adoption.

We conclude that it is possible to enforce full spatial and temporal safety with low performance overheads using pointer-based checking with disjoint metadata on legacy C programs. Efficient streamlined implementation is crucial in obtaining low performance overheads as we demonstrated it with *SoftBoundCETS*, *Watchdog* and *HCH*. From our experience in building pointer-based checking in various parts of the tool chain, we conclude that the time is ripe for lightweight *HCH* like acceleration that reduces the performance overheads significantly while minimizing the hardware investment with the help of the compiler.

8.2 Impact and Adoption

In the time since we presented our initial research papers that laid the groundwork for this dissertation at PLDI 2009, ISMM 2010 and ISCA 2012, Intel has announced support for pointer-based checking with disjoint metadata in its latest C/C++ production compiler [57]. Like the *SoftBound-CETS* instrumentation proposed in this dissertation, Intel's tool is also a compiler-based implementation of pointer-based checking with per-pointer disjoint metadata. The motivation (security & debugging) and the description of the tool's operation (how bounds information is calculated and propagated) are similar to what was described in our PLDI paper. Further, Intel's tool uses the same solutions proposed in our research to narrow the bounds of the pointers pointing to internal objects.

Although Intel's tool is primarily focused on detecting bounds errors, it provides some protection against dangling pointer errors by changing the bounds of the freed pointer. However, to provide comprehensive protection, such a scheme would need to change the bounds of all pointers aliasing with the freed pointers in memory or can adopt the identifier-based checking used in this dissertation. The Intel tool has a higher overhead than the spatial safety mode of *SoftBound-CETS* (our spatial safety scheme has 74% overhead in contrast to Intel's tool that has 500% or more overhead), probably because the current Intel tool is focused on debugging and needs to meet the robustness constraints of an actual product.

To summarize, this industrial adoption of the specific approach used in this dissertation supports its overall feasibility as an effective and practical technique for preventing memory safety violations. However, Intel's compiler-based instrumentation is currently too slow (5$\times$ or more slowdown) to use for production code. Although Intel has not announced any hardware support for it, the argument for adding such support now becomes a simpler argument for accelerating an already-established checking paradigm.

8.3 Future Directions and Further Challenges

In this dissertation, we focused on providing comprehensive memory safety for legacy C programs with low performance overheads. We highlight some of the avenues that we did not explore in this dissertation but are attractive for wide adoption of memory safety enforcement techniques for C programs.

How Can Static Optimizations Help? We think that there are three directions where static optimizations can significantly help the pointer-based checking approach proposed in this dissertation. First, we did not explore many optimizations especially on the spatial checking front. Check optimizations with sophisticated analyses can not only eliminate the checks [23, 62, 130] but also metadata accesses that feed just the checks. As we currently do not perform any loop peeling-based spatial check elimination, there is an opportunity to design custom spatial check elimination that can eliminate spatial checks [130]. Improving the precision of the alias analysis, custom check elimination, and domain specific information on eliminating checks will further reduce the overhead of pointer-based checking. Second, one of the key features that makes checking for C hard is the conflation of pointers to unary objects (singletons) and arrays. Data structure and shape analysis that can infer singletons data objects precisely are likely to benefit spatial checking. Third, a significant direction that we did not explore fully in this dissertation is the use of type adherence to enforce memory safety for C programs. In this dissertation, we adopted pointer-based checking as we wanted to enforce memory safety for legacy programs. If we change our goal from legacy programs to future programs, then creating a subset of C or its variants to enforce type safety while being expressive is an attractive avenue. In a collaborative effort, we are currently exploring avenues to subset C++, a variant of C, that eliminates most unsafe casts to provide type adherence, which assists in providing memory safety.

How to Leverage and Provide Programmer Guidance? We did not leverage any programmer provide annotations in this dissertation. Designing annotations that can incorporate programmer feedback in the process of enforcing memory safety is an attractive avenue. Current language extensions and annotation languages [73] take an all or nothing approach where a programmer is required to annotate the program without which the system does not work. An incremental approach that includes a default full memory safety scheme coupled with a system that leverages programmer provided annotations would further reduce the overheads and make pointer-based checking attractive for wider adoption.

Further, we did not provide guidance to the programmer about performance overheads. Although a program having a large number of metadata loads/store is indicative of performance overheads, providing concrete feedback about regions that are experiencing performance overheads due to memory safety checks would be useful. Identification of such regions would enable programmers to either restructure their code or provide annotations to reduce the overhead. There are many such opportunities to restructure the code in existing applications. For example, when we experimented with *equake*, we were seeing high overheads (300-400%) with *SoftBoundCETS*. We observed that the benchmark was performing a huge number of metadata loads and stores. Further exploration enabled us to figure out that the bad design choice in use of multidimensional arrays was the reason for the overheads. We modified a few lines in the program to use a proper multidimensional array that improved the performance of the baseline (without memory safety instrumentation) by 60%. Further, it reduced the performance overhead with *SoftBoundCETS* to around 50%.

What are the Performance Implications with Multi-threaded Code? In this dissertation, we focused our attention on enforcing memory safety for single-threaded programs. We discussed the issues that we had to address to provide memory safety for multi-threaded programs in this dissertation (Section 6.7). Understanding the performance implications of various design choices in a multi-threaded setting will be interesting. Further, new optimizations probably need to be designed to eliminate checks in a multithreaded context.

Beyond the future research directions, the goal of this dissertation will come to fruition only when the broader community embraces these ideas and memory safe C programs become a reality.

8.4 Reflections on Memory Safety

This section offers my thoughts on the future of C programming language, its variants and enforcing memory safety for them.[1] These are opinions that I developed working on this dissertation for the last five years. These opinions have also been shaped by my experience with real-world tools, other research in this area, interactions with other safety/security researchers and the user community.

C and its Variants Are Not Going Away Anytime Soon Although many managed language have emerged and have become popular, the key attribute that distinguishes C (and its variants) from the rest is its focus on performance. On the other hand, some of the nice aspects of managed languages are its focus on programmer productivity and safety. To bridge this gap, the C programming language and its variants have evolved over time. When C and its variants provide type adherence, and

[1] These are my own thoughts and opinions and not necessarily shared by my co-authors. I use the singular pronoun throughout this section.

are bestowed with a lightweight runtime that enforces memory safety, the difference in productivity between the managed world and the C world narrows down significantly, but the performance advantages may remain. Then, the future C world with a nice ecosystem of libraries and utilities combined with its focus on performance may become the ecosystem of choice rather than need.

Enforcing Spatial Safety For Type Safe Programs Is Not Hard Among the two concerns in enforcing memory safety: spatial and temporal safety, enforcing spatial safety for programs that are type safe is not difficult in agreement with the observations by CCured [94]. When the type casts are eliminated from C programs or its variants, then a pointer-based checking scheme just needs to check array accesses. Moreover, there is no need for disjoint metadata. Further, my experience with C code indicates that programmers use these type casts to create poor man's object-oriented paradigms in C programs. An object oriented variant of C, *i.e.* C++, which provides first class support for object-oriented programming often eliminates such casts. Spatial safety for such type-safe variants can be enforced with a low overhead.

Temporal Safety vs Garbage Collection Garbage collection has emerged as an alternative to manual memory management. Garbage collection address two issues compared to manual memory management: (1) increases programmer productivity as programmers do not have to manually manage memory, and (2) avoids temporal safety violations that result with the incorrect use of manual memory management. Garbage collection may be an attractive option for domains where either the programs run for a short duration of time or the programs are not performance sensitive. A variant of C that is type-safe can benefit from the research in the garbage collection community to build a precise garbage collector.

On the other hand, for programs where garbage collection is not an option, providing mechanisms to check that the programmers manage memory correctly is important. The use of temporal safety mechanisms proposed in this dissertation will be attractive in those domains. In contrast to spatial safety checking, unfortunately, enforcing temporal safety for type-safe programs is not appreciably easier than it is for non-type safe programs.

Separate Compilation/Libraries and Debugging Support From my experience in developing and maintaining *SoftBoundCETS* in the LLVM experimental subversion repository, supporting separate compilation and maintaining library compatibility is important. Many of the widely used utilities have esoteric build environments that generally use separate compilation. Enabling memory safety checking using a push-button methodology with a compile time flag would foster adoption avoiding the inertia in switching tools.

Ideally one would like to recompile all libraries to enforce memory safety. In the absence of such recompilation, wrappers are required to interface with the libraries. Although writing wrappers is a one-time effort, my experience writing it for a wide range of libraries indicates that it is still hard even with disjoint metadata. To increase adoption, recompilation of libraries is essential.

An important aspect that is often overlooked in building memory safety enforcement tools is providing good debugging support. Many of the memory safety tools either overlook debugging support in the hunt for lower performance overheads or neglect performance overheads being primarily debugging tools. Providing good debugging support with low performance overheads is crucial for programmers to not only detect these bugs but also provide them enough contextual information to fix them.

Misconceived Notions About Overheads Over the last five years, I have talked to numerous developers to encourage the wide adoption of memory safety checking tools. I realized that many developers have misconceived notions of performance overhead. Many developers want tools to have zero performance impact for programs that they are unwilling to change. In my experience, I have observed that most programs can be tuned with minor changes that can reduce the performance overhead of memory safety checking substantially. Further, my experience in tuning *equake* demonstrates that there are opportunities to restructure and tune the existing C programs that not only reduces the overheads with memory safety checking but also improves the performance of the baseline. In my opinion, overheads in the range of 40-50% are largely unnoticeable for many programs except for hand-crafted highly tuned programs. Many developers even disable advanced compiler optimizations which itself slows down the program more than these reported overheads. For highly tuned programs, providing annotations and having a feedback loop similar to annotations with approaches like Deputy [35, 73] are attractive. Building interactive tools that check the validity of the annotations are equally important as the memory safety checking tools themselves to make C programs memory safe.

Bibliography

[1] Electric Fence. URL http://perens.com/FreeSoftware/ElectricFence/.

[2] Google Hack Attack Was Ultra Sophisticated, New Details Show. URL http://www.wired.com/threatlevel/2010/01/operation-aurora/.

[3] Heap Spray Exploit Tutorial: Internet Explorer Use After Free Aurora Vulnerability. URL http://grey-corner.blogspot.com/2010/01/heap-spray-exploit-tutorial-internet.html.

[4] National Vulnerability Database. NIST. URL http://web.nvd.nist.gov/.

[5] PageHeap. URL http://msdn.microsoft.com/en-us/library/cc265936.aspx.

[6] SoftBound website. URL http://www.cis.upenn.edu/acg/softbound/.

[7] Adobe Reader vulnerability exploited in the wild, 2008. URL http://www.trustedsource.org/blog/118/Recent-Adobe-Reader-vulnerability-exploited-in-the-wild.

[8] Adobe Security Advisories: APSB08-19, Nov. 2008. URL http://www.adobe.com/support/security/bulletins/apsb08-19.html.

[9] Dangling pointer vulnerability in nsPluginArray, 2008. URL http://www.mozilla.org/security/announce/2010/mfsa2010-19.html.

[10] Use-after-free vulnerability when viewing XUL document with script disabled, 2011. URL http://www.mozilla.org/security/announce/2011/mfsa2011-20.html.

[11] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, Nov. 2005.

[12] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX conference on Security*, 2010.

[13] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[14] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium*, Aug. 2009.

[15] D. Arora, A. Raghunathan, S. Ravi, and N. K. Jha. Architectural Support for Safe Software Execution on Embedded Processors. In *Proceedings of the International Conference on Hardware Software Co-design and System Synthesis*, Oct. 2006.

[16] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994.

[17] H. Aygn. *D.U.M.A. — Detect Unintended Memory Access*. URL `http://duma.sourceforge.net/`.

[18] A. Baratloo, N. Singh, and T. Tsai. Transparent Runtime Defense Against Stack Smashing Attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[19] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Inject Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[20] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[21] S. Bhatkar and R. Sekar. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[22] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[23] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[24] H.-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993.

[25] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data Randomization. Technical report, Microsoft Research, 2008.

[26] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical report, The UPC Consortium, 1999.

[27] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[28] J. M. Chang and E. F. Gehringer. Evaluation of an object-caching co-processor design for object-oriented systems. In *Proceedings of the International Conference on Computer Design*, Oct. 1993.

[29] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008.

[30] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008.

[31] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2005.

[32] W. Chuang, S. Narayanasamy, and B. Calder. Accelerating Meta Data Checks for Software Correctness and Security. *Journal of Instruction-Level Parallelism*, 9, June 2007.

[33] W. Chuang, S. Narayanasamy, and B. Calder. Bounds Checking with Taint-Based Analysis. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, Jan. 2007.

[34] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm, 2005.

[35] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent Types for Low-Level Programming. In *Proceedings of the 16th European Symposium on Programming*, Apr. 2007.

[36] K. D. Cooper, M. W. Hall, and K. Kennedy. A Methodology for Procedure Cloning. *Comput. Lang.*, 19(2):105–117, 1993.

[37] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[38] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the Foundations of Intrusion Tolerant Systems*, 2003.

[39] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[40] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[41] R. Cytron, J. Ferrante, B. K. Rosen, M. k. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[42] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[43] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[44] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[45] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2006.

[46] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[47] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES)*, 2003.

[48] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2004.

[49] W. Dormann. *Microsoft Internet Explorer Data Binding Memory Corruption Vulnerability*. US-CERT, Dec. 2008. URL `http://www.kb.cert.org/vuls/id/493881`.

[50] M. Drinic and D. Kirovski. A Hardware-Software Platform for Intrusion Prevention. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[51] F. C. Eigler. Mudflap: Pointer Use Checking for C/C++. In *GCC Developer's Summit*, 2003.

[52] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[53] D. Evans. Static Detection of Dynamic Memory Safety Errors. In *Proceedings of the SIGPLAN 1996 Conference on Programming Language Design and Implementation*, May 1996.

[54] A. M. Fiskiran and R. B. Lee. Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution. In *Proceedings of the International Conference on Computer Design*, Oct. 2004.

[55] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implementation*, June 1998.

[56] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[57] K. Ganesh. *Pointer Checker: Easily Catch Out-of-Bounds Memory Accesses*. Intel Corporation, 2012. URL `http://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf`.

[58] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implementation*, June 1998.

[59] D. Gay, R. Ennals, and E. Brewer. Safe Manual Memory Management. In *Proceedings of the 2007 International Symposium on Memory Management*, Oct. 2007.

[60] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman. Architectural Support for Low Overhead Detection of Memory Viloations. In *Proceedings of the Design, Automation and Test in Europe*, 2009.

[61] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[62] R. Gupta. A Fresh Look at Optimizing Array Bound Checking. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990.

[63] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular Checking for Buffer Overflows in the Large. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[64] T. Harris, S. Tomic, A. Cristal, and O. Unsal. Dynamic filtering: multi-purpose architecture support for language runtime systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[65] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of the Winter Usenix Conference*, 1992.

[66] M. Hertz and E. D. Berger. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. 2005.

[67] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience With Safe Manual Memory Management in Cyclone. In *Proceedings of the 2004 International Symposium on Memory Management*, Oct. 2004.

[68] A. Hilton and A. Roth. BOLT: Energy-Efficient Out-of-Order Latency-Tolerant Execution. In *Proceedings of the 16th Symposium on High-Performance Computer Architecture*, Feb. 2010.

[69] A. Hilton and A. Roth. Decoupled Store Completion/Silent Deterministic Replay: Enabling Scalable Data Memory for CPR/CFP Processors BOLT: Energy-Efficient Out-of-

Order Latency-Tolerant Execution. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010.

[70] A. D. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating All-Level Cache Misses in In-Order Processors. In *Proceedings of the 15th Symposium on High-Performance Computer Architecture*, Feb. 2009.

[71] *Microsoft Internet Explorer HTML TIME 'ondatasetcomplete' Use After Free Vulnerability*. iDefense Labs, June 2009. URL `http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=817`.

[72] *Multiple Vendor WebKit Error Handling Use After Free Vulnerability*. iDefense Labs, June 2009. URL `http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=803`.

[73] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[74] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[75] R. W. M. Jones and P. H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Third International Workshop on Automated Debugging*, Nov. 1997.

[76] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1998.

[77] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[78] S. C. Kendall. BCC: Runtime Checking for C Programs. In *Proceedings of the USENIX Summer Technical Conference*, 1983.

[79] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and Efficiently Protecting the Heap. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[80] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.

[81] G. Kroah-Hartman. The Linux Kernel Driver Model: The Benefits of Working Together. In A. Oram and G. Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, Inc., June 2007.

[82] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[83] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[84] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, June 2005.

[85] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[86] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[87] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for Evaluating Bug Detection tools. In *In PLDI Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[88] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[89] M. Meyer. A Novel Processor Architecture with Tag-Free Pointers. In *IEEE Micro*, 2004.

[90] G. Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[91] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, June 2012.

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

[93] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, June 2010.

[94] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[95] N. Nethercote and J. Fitzhardinge. Bounds-Checking Entire Programs Without Recompiling. In *Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2004.

[96] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2007.

[97] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[98] *NIST Juliet Test Suite for C/C++*. NIST, 2010. URL `http://samate.nist.gov/SRD/testCases/suites/Juliet-2010-12.c.cpp.zip`.

[99] G. Novark and E. D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[100] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[101] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[102] Y. Oiwa. Implementation of the Memory-safe Full ANSI-C Compiler. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, June 2009.

[103] H. Patil and C. N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software — Practice & Experience*, 27(1):87–110, 1997.

[104] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: protecting critical data in unsafe languages. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.

[105] C. Peng and G. S. Sohi. Cache Memory Design Considerations to Support Languages with Dynamic Heap Allocation. Technical report, University of Wisconsin-Madison, 1989.

[106] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[107] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[108] P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. Technical report, SRI International, Feb. 2009.

[109] F. Qin, Z. Li, Y. Zhou, C. Wang, H. Kim, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006.

[110] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, Feb. 2005.

[111] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: a defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium*, 2009.

[112] A. Roth. Physical Register Reference Counting. *IEEE TCCA Computer Architecture Letters*, 7(1), Jan. 2008.

[113] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed Systems Security Symposium*, Feb. 2004.

[114] W. J. Schmidt and K. D. Nilsen. Performance of Hardware-Assisted Real-Time Garbage Collector. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[115] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[116] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Apr. 2005.

[117] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[118] W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, and J. Yang. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[119] W. Shi and H.-H. S. Lee. Authentication Control Point and its Implications for Secure Processor Design. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006.

[120] M. S. Simpson and R. K. Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *IEEE International Workshop on Source Code Analysis and Manipulation*, 2010.

[121] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: preventing buffer overflows for recompilation, 2012.

[122] G. Smith and D. Volpano. A Sound Polymorphic Type System for a Dialect of C. *Science of Computer Programming*, 1998.

[123] J. L. Steffen. Adding Runtime Checking to the Portable C Compiler. *Software Practice and Experience*, 22(4):305–316, 1992.

[124] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[125] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[126] G. Venkataramani, B. Roemer, M. Prvulovic, and Y. Solihin. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.

[127] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2000.

[128] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th Symposium on Operating System Principles*, Oct. 1993.

[129] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[130] T. Würthinger, C. Wimmer, and H. Mössenböck. Array Bounds Check Elimination for the Java HotSpot Client Compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, 2007.

[131] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[132] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. NativeClient: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

[133] S. H. Yong and S. Horwitz. Protecting C Programs From Attacks via Invalid Pointer Dereferences. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2003.

[134] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communication, Computer and Communications Security*, 2010.

[135] Q. Zeng, D. Wu, and P. Liu. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the SIGPLAN 2011 Conference on Programming Language Design and Implementation*, June 2011.

[136] K. Zetter. Operation Aurora. URL `http://en.wikipedia.org/wiki/Operation_Aurora`.

[137] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of The 39th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 2012.

[138] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and Scalable Memory Shadowing. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2010.

[139] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[140] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-related Bugs Via Program Counter-based Invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[141] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[142] X. Zhuang, T. Zhang, and S. Pande. Using Branch Correlation to Identify Infeasible Paths for Anomaly Detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006.