HardBound: Architectural Support for Spatial Safety of the C Programming Language

Joe Devietti [*]
University of Washington
devietti@cs.washington.edu

Colin Blundell
University of Pennsylvania
blundell@cis.upenn.edu

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

The C programming language is at least as well known for its absence of spatial memory safety guarantees (*i.e.*, lack of bounds checking) as it is for its high performance. C's unchecked pointer arithmetic and array indexing allow simple programming mistakes to lead to erroneous executions, silent data corruption, and security vulnerabilities. Many prior proposals have tackled enforcing spatial safety in C programs by checking pointer and array accesses. However, existing software-only proposals have significant drawbacks that may prevent wide adoption, including: unacceptably high runtime overheads, lack of completeness, incompatible pointer representations, or need for non-trivial changes to existing C source code and compiler infrastructure.

Inspired by the promise of these software-only approaches, this paper proposes a *hardware bounded pointer* architectural primitive that supports cooperative hardware/software enforcement of spatial memory safety for C programs. This bounded pointer is a new hardware primitive datatype for pointers that leaves the standard C pointer representation intact, but augments it with bounds information maintained separately and invisibly by the hardware. The bounds are initialized by the software, and they are then propagated and enforced transparently by the hardware, which automatically checks a pointer's bounds before it is dereferenced. One mode of use requires instrumenting only `malloc`, which enables enforcement of per-allocation spatial safety for heap-allocated objects for existing binaries. When combined with simple intra-procedural compiler instrumentation, hardware bounded pointers enable a low-overhead approach for enforcing complete spatial memory safety in unmodified C programs.

Categories and Subject Descriptors C.0 [*Processor Architectures*]: Hardware/software interfaces; D.2.0 [*Software Engineering*]: General—Protection mechanisms; D.3.4 [*Processors*]: Memory management

General Terms Languages, Security, Performance

Keywords Spatial memory safety; C programming language

1. Introduction

The C programming language is the de facto standard for systems programming, and software written in C (or its sibling C++) makes up the majority of code running on most platforms. This success is due in part to the low-level control over data representation, memory management, and performance that C gives programmers. Despite this widespread use, there is a price to pay: C is the source of a range of software vulnerabilities that permeate our computing infrastructure. The root of the problem is that the C language is inherently unsafe. Its unchecked array operations lead to buffer overflows; the conflation of pointers and arrays allows hazardous pointer arithmetic and dereferencing; unsafe casts allow programs to accidentally write to or read from arbitrary memory addresses.

[*] This work done while Joe Devietti was at the University of Pennsylvania.

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/03…$5.00

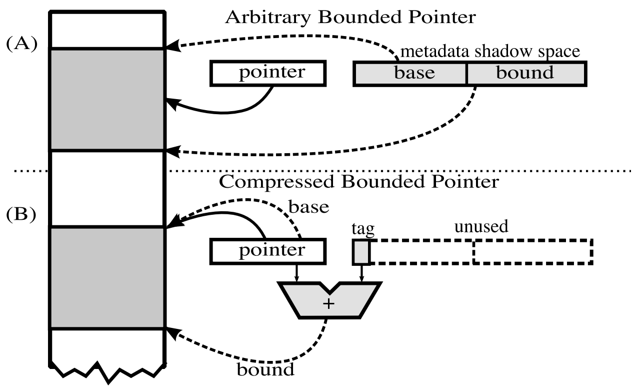

Figure 1. Bounded pointers: `(A)` Using full base/bound metadata and `(B)` Compressed (pointer equals base and the object is small).

There have been many proposals that ameliorate the problems caused by C's unchecked pointer and array accesses by partially or fully detecting violations of *spatial memory safety*. A violation of spatial memory safety occurs when a program uses a variable to access memory that is outside the bounds of the object associated with the variable. Spatial errors include accessing the *n*th element of an *m*-element array when $n > m$, erroneously indexing off a non-array pointer, or casting a pointer to a `struct` larger than the region originally allocated and then accessing a field that is beyond the bounds of the original allocation.

To help detect and diagnose spatial errors in C programs, many software-only tools (*e.g.*, [3, 19, 24, 42, 43, 44, 50]) and hardware-supported techniques (*e.g.*, [32, 47, 59, 65]) have been proposed. Although these techniques are useful, many of them do not provide complete spatial memory safety. Likewise, many special-purpose techniques (in software and hardware) address restricted classes of security exploits made possible by spatial memory safety violations. These approaches focus on protecting the return address [9, 34, 39], protecting data pointers [8] or code pointers [57], detecting anomalous program flow [20], protecting heap metadata [29], or preventing memory attacks by tracking untrusted inputs via tainting [10, 46, 55, 58]. Although effective in many cases, these targeted proposals mostly focus on specific attacks or symptoms and not on the root cause of the problem.

Instead of relying on this patchwork of incomplete and indirect solutions, other approaches have directly attacked the source of the problem: C's lack of spatial memory safety. Just as type-safe languages like Java and C# eliminate all of the vulnerabilities mentioned above, an implementation of C that enforces spatial safety will also avoid them. Several promising software-only approaches for enforcing full or almost-full spatial safety for C have been proposed (*e.g.*, [2, 7, 14, 27, 28, 40, 45, 49, 61, 62]).

Unfortunately, these software-only proposals all suffer from one or more deficiencies that may prevent wide adoption, such as: unacceptably high runtime overheads, incomplete detection of spatial violations, incompatible pointer representations (by changing memory layout), or requiring non-trivial changes to existing C source code. Moreover, the software-only schemes with the lowest performance overheads generally require sophisticated whole-program compiler analyses (*e.g.*, [14, 40]). Section 2 discusses these software techniques in detail.

This paper describes HardBound, a new hardware design that overcomes the deficiencies of these software-only approaches by providing architectural support for a new primitive datatype—a *hardware bounded pointer*—inspired by the pointers used in Safe-C [2], CCured [40], and Cyclone [27]. These software-based schemes replace some or all of the pointers in the program with three-word "fat" pointers that encode the actual pointer, the base address of the associated object, and its bound (as illustrated in Figure 1(A)). Unlike the purely software approaches to implementing fat pointers, our proposed HardBound support (1) maintains memory layout compatibility by encoding the bounds information in a disjoint shadow space, (2) implicitly checks and propagates the bounds information as the bounded pointer is dereferenced, incremented, and copied to and from memory, and (3) reduces storage and runtime overheads by caching compressed pointer encodings, thereby allowing many bounded pointers to be efficiently represented using just a few additional bits of state (as illustrated in Figure 1(B)).

Hardware bounded pointers are intended to facilitate software enforcement of spatial memory safety—the software is responsible for communicating valid bounds metadata to the hardware via calls to a new `setbound` instruction. This design permits flexible use of HardBound primitives, ranging from simple bounds protection at the heap-allocated object granularity (which requires only minor changes to `malloc()` and is binary-compatible with legacy code) to CCured-style complete spatial safety.

To summarize, this paper makes the following contributions:

- We describe HardBound—a hardware bounded pointer primitive—and accompanying compiler transformations that together enforce complete spatial safety for C programs. Section 3 describes the hardware bounded pointer model, hardware bounds propagation, and their use for spatial safety. The HardBound approach strives to minimize changes to the compiler infrastructure, and it retains compatibility with legacy C code with respect to memory layout.
- We propose an efficient implementation of hardware bounded pointers (in Section 4) that opportunistically uses a compressed metadata encoding. In the uncommon case, the full base and bound metadata are stored in a reserved portion of virtual memory. In the common case of pointers to small objects and non-pointer data, the hardware encodes the bounded pointer metadata using just a few bits. These bits are stored either in memory or in unused bits of the pointer itself. In both cases, the hardware performs the encoding and decoding, making the specific encoding transparent to the software.
- We experimentally evaluate both the functional correctness and performance of our approach (in Section 5). HardBound accurately detects all spatial memory violations in an extensive suite of spatial violation test cases [31]—with no false positives. Performance measurements of a simulated x86 processor on a variety of benchmarks indicate that the runtime overhead is just 5% to 9% on average depending on the pointer encoding.

Although spatial safety enforcement eliminates a large class of bugs and security vulnerabilities, it does not eliminate all of them. As discussed in Section 6.1, HardBound provides just enough type safety to enforce full spatial safety, but it does not provide full type safety. HardBound also does not address temporal memory safety errors (*e.g.*, dangling pointers and uninitialized memory reads). Section 6.2 considers temporal-safety issues and suggests how HardBound may be used in conjunction with existing temporal-safety protection mechanisms.

Before describing our hardware bounded pointers, we first overview the prior software-only approaches for detecting spatial memory violations in C that motivated and inspired this work.

2. Background: Detecting Spatial Violations in C

Detecting spatial safety violations in C programs is not a new problem. Several techniques for detecting spatial memory violations for C were proposed in the 1990s as debugging aids, and more recent work has improved efficiency to the point where they are arguably fast enough for everyday use. The next few subsections describe and compare these approaches, focusing on their performance, completeness, and compatibility attributes. Because our focus is on spatial violations, we defer discussion of the temporal violation detection aspects of these approaches until Section 6.2. Proposals that focus on information flow, taint analysis, or tamper-resistant hardware are discussed later in Section 7.

2.1 Red-Zone Tripwire Approaches

One approach to detecting spatial violations is to track a few bits of state for each byte in memory; the additional bits indicate whether the location is currently valid [24, 43, 47, 59, 62]. As memory is allocated, the bytes are marked as valid. Every load or store is instrumented to check the validity of the location. By placing a "red-zone" block of invalid memory between memory objects, contiguous overflows—caused by walking past an array boundary with small stride—will hit the red-zone tripwire, assuming the red-zone size is larger than the stride. These techniques are not complete: large overflows may jump over the tripwire and access data from another object undetected, causing a spatial safety violation.

Purify [24] and Valgrind's MemCheck [43] implement the tripwire approach using binary rewriting, but their large performance overheads restrict them to use only during software development. Yong *et al.* [62] use static analysis and check only memory writes to reduce the runtime overhead of this approach to under 2x in many cases. The overheads of this technique can be further reduced by either using invalid ECC signatures to encode invalid memory locations [47] or adding hardware support for updating and checking the valid/invalid blocks [59]. Although useful for finding spatial violations (and many temporal errors, as discussed in Section 7), a significant drawback of these schemes is that they cannot guarantee the detection of all spatial violations.

2.2 Object Lookup Approaches

The second general approach is to track the size of each object in a separate data structure and ensure that all pointer arithmetic and pointer dereferences fall within the bounds of the original object [28]. Because legal C programs are allowed to increment a pointer past the end of an object, an error should be triggered only when an out-of-bounds pointer is actually dereferenced. To distinguish between an out-of-bounds pointer and a pointer to the next object in memory, such a pointer is changed to point to a special out-of-bounds object [49]. If later pointer arithmetic puts the pointer back in bounds, the pointer must be correctly restored. The object lookup table is typically implemented as a splay tree in which objects are identified with their locations in memory, yielding runtime overheads of 5x [28]. Optimizations can reduce overhead by improving the implementation [14, 19, 49], checking only strings [49], caching tree lookups [14, 19], or using static analysis to elide tracking of non-array objects and to enable multiple splay trees [14].

The most important advantage of this approach is that the layout of objects in memory is unchanged, which provides fewer source and binary compatibility issues than the fat pointer schemes described below. Unfortunately, the object lookup approach can suffer from high runtime overheads unless combined with sophisticated whole program analysis [11, 14]. Another disadvantage is that this approach cannot detect all spatial violations because the bounds of arrays inside `structs` are not checked [14, 28]. For example:

```
struct {char str[5]; int x;} node;
char *ptr = node.str;
strcpy(ptr, "overflow"); // overwrite node.x
```

With the above code, pointers to `node` and `node.str` are indistinguishable (they are the same address). Because both pointers map to a single table entry, a pointer to `node.str` is given bounds of the whole `node` object. Thus, when `ptr` is passed to `strcpy()`—even if `strcpy()` has been instrumented—an overflow of `node.str` that does not overflow the entire structure will not be detected. As `node.x` could have been a data or function pointer, this undetected spatial memory violation could lead to a serious memory corruption bug or security vulnerability.

2.3 Fat Pointer Approaches

The third general approach is to use a fat pointer representation that replaces some or all pointers with multi-word pointer/base/bound triples as shown in Figure 1(A). These three-word bounded pointers represent the actual pointer value together with the addresses of the upper and lower bounds of the object. A bounded pointer can be incremented (by changing just its value portion) and yet still be verified to be within the array (by checking the value against the two bounds). If the pointer is incremented too far, any out-of-bounds access will be detected when the pointer is dereferenced. This strategy avoids the problems with object-indexed tables, because multiple pointers to the same base address can be given different bounds. Proposals such as SafeC [2], CCured [40], Cyclone [23, 26, 27], and others [32, 42, 45, 61] use fat pointers to enforce spatial safety by checking that every pointer dereference falls between its associated base and bound.

The primary advantage of fat pointers is that they can be used to enforce complete spatial safety. However, propagating and checking bounds for all pointers can result in 2x or more runtime overhead [2]. Consequently, various techniques have been proposed to safely eliminate the use of fat pointers. Cyclone, for example, is a C-like language that explicitly distinguishes non-array pointers from array pointers. Non-array pointer bounds are validated by static typechecking, so only array pointers are required to be fat. Compared with unsafe C programs, Cyclone's dynamic checks increase runtime by about 40% on a range of benchmarks [27]. Cyclone's primary drawback is that it requires significant effort to port C programs—Cyclone is a new language.

CCured [40] uses whole-program type inference to statically optimize the use of different kinds of pointers, trading off between the performance overheads and the degree of flexibility in their use. `SAFE` pointers have almost no overhead, but cannot be used for pointer arithmetic, array indexing, or type casts. `SEQ` pointers are fat pointers that allow pointer arithmetic, array indexing, and some casts. To support arbitrary casts, CCured uses `WILD` pointers, which require expensive dynamic checks and additional metadata. CCured's type inference dramatically reduces run-time costs associated with safety checks as compared to Safe-C, but the overheads can still be significant: the CCured papers report execution time overheads of 3%–87% on a range of benchmarks.

2.4 Analysis and Comparison

The object table and fat pointer schemes have complementary strengths and weakness. Object table approaches are highly-compatible as they avoid changing memory layout—so compatible they have been successfully applied to the entire Linux kernel [11]—but they do not enforce complete spatial safety. Fat pointer approaches can enforce complete spatial safety, but the memory layout and pointer representation changes cause source code and library incompatibilities [7, 40, 61, 64]. Attempts have been made to mitigate the compatibility issues of fat pointers by splitting out the bounds and base metadata (*e.g.*, [40, 61]), but such techniques can result in shadow structures that mirror entire existing data structures. Even with such splitting support, the CCured developers marked some program statements as trusted to avoid creating `WILD` pointers (and the significant performance issues caused by them [40]). Deputy [7, 64], a follow-on project to CCured, ensures spatial safety at runtime while avoiding fat pointers. To accomplish this, it uses dependent type annotations to associate pointers with bounds metadata already present in the program. This approach mitigates the memory layout compatibility issues at the cost of programmer-inserted annotations.

Another potential concern is that the most efficient implementations of both object table and fat pointer methods use whole-program analysis. Although whole-program analysis is becoming more widely used, it is currently not commonly used for C programs. Furthermore, precompiled libraries and dynamically loaded code can significantly limit such analysis. The inference algorithms used by these implementations also have the property that a small change in one part of the program (*e.g.*, use of pointer arithmetic or type cast) can have a significant impact on the runtime of other seemingly unrelated parts of the program. Such effects make it difficult for programmers to reason about performance. Finally, with these innovations the runtime overheads are acceptable on average, but some benchmarks still incur significant runtime overheads.

In summary, the fat pointer approach is appealing because it can enforce complete spatial safety, but it suffers from compatibility issues as well as high runtime overheads in some cases. Recent proposals have demonstrated that these performance overheads can be lowered by adding bounds checking instructions [1, 5], but the compatibility issues remain. In the next section, we describe HardBound, our proposal that makes the fat pointer approach binary compatible as well as providing increased performance over software-only schemes.

3. A Hardware/Software Approach

HardBound's goal is to provide a hardware primitive that allows a C compiler to enforce the complete spatial memory safety of the fat pointer approach, retain the binary compatibility of the object-table approach, and incur lower overheads than the fastest implementation of either. HardBound thus provides ISA support for first-class bounded pointers that are intended to meet the following criteria:

- **Completeness:** It should be possible to use HardBound's primitives to enforce spatial safety guarantees as strong as CCured's.
- **Performance:** The hardware should yield performance comparable to or better than the best performing software-only approaches.
- **Binary compatibility:** The metadata needed for HardBound's bounded pointers should be transparent to the source program so that legacy data layout, library interfaces, and legacy code compatibility are preserved.
- **Source compatibility:** HardBound should be usable with minimal modifications to existing C source code.

- **Minimal compiler support:** Using HardBound should not require compilers that do whole-program analysis or require extensive modifications to the runtime system.

One important consideration is the division of labor between the compiler and hardware. In HardBound's design, the compiler and/or runtime system is responsible for creating bounded pointers by (1) communicating initial base and bounds information to the hardware and (2) occasionally tightening pointer bounds (*e.g.*, when the program creates a pointer to a substructure). The hardware is responsible for (1) dynamically checking that all memory accesses are within the specified bounds and (2) propagating the metadata as the pointer is manipulated in memory and registers (*e.g.*, when a pointer is copied or incremented).

In HardBound, as in software-only approaches, the compiler guarantees memory safety—an incorrect compiler implementation may produce unsafe binary code. However, by making fat pointers cheap, HardBound reduces the need to do whole-program analysis, simplifying the compiler. Section 3.2 describes how the HardBound primitives can be used to enforce spatial safety, but we first give a high-level description of HardBound's bounded pointer datatype (Section 3.1). The discussion of efficient implementations of HardBound is deferred to Section 4.

3.1 A Bounded Pointer Hardware Primitive

HardBound (conceptually) extends every register and word of memory in the virtual address space with a "sidecar" shadow base and bound. Instead of being single values, the architected state of registers and memory locations are now triples `{value; base; bound}`. The *base* address is the first valid address of the region; the *bound* is the first address after the end of the region (see Figure 1). For non-pointer values, the base and bound portion are set to zero and ignored. For pointer values, the base and bound are used to perform an implicit bounds check for every load or store operation. Storing the base and bound information in sidecar shadow spaces has the advantage of not changing the program's view of the memory layout of datatypes, and it simultaneously allows for an efficient compressed encoding of the bounds information in the common case (see Section 4).

Setting and propagating bounds information in registers. The hardware provides a `setbound` instruction that adds or modifies the bounds information of a pointer.[1] The `setbound` instruction takes an input register that contains a memory address and an input register (or immediate) with the size of the region to which the pointer will be bounded. For example, lines 1 and 2 in Figure 2 create a bounded pointer to an array of size four that begins at memory address `0x1000`. Such code might be executed within a `malloc` invocation that performs a four-byte allocation.

Whenever the processor performs a pointer dereference, the effective address of the pointer is checked to be between its associated base and bound. This check occurs implicitly as part of every load or store operation to memory. If the bounds check passes, no action is taken; if the check fails, the processor raises an exception. The runtime system handles the exception by either terminating the process or invoking some other language-specific exception. Continuing the example in Figure 2, the `load` instruction on line 3 passes the bounds test associated with the bounded pointer in `R2`; the `load` in line 4 fails.

Pointer arithmetic and other pointer manipulations are common in C programs. To free the compiler from the burden of explicitly maintaining and propagating bounds information (and eliminate the associated run-time overhead), the hardware automatically propagates the bounds information when a register containing a pointer is manipulated. For example, when an offset is added to a pointer, the destination register inherits the same bounds information as the original pointer. Line 5 of Figure 2 shows the result of incrementing the pointer in `R2` and storing the result in `R5`—although the value component is incremented, the base and bound components are copied unchanged.

Whether the output of an instruction inherits bounds information is determined by the specific operation and pointer/non-pointer status of the input registers. For example, adding a pointer to an immediate or another non-pointer register propagates the bounds from the pointer register. As such, a word-sized register-register addition instruction would be defined as in Figure 3 (A) and (B).

Any instruction that directly manipulates pointers propagates the pointer information in this way. For example, subtracting a value from a pointer also propagates the bounds, as do these other instructions (in the x86 ISA): `add`, `sub`, `lea`, `mov`, and `xchg`. For other operations that are not typically used to calculate pointers (multiply, divide, shift, rotate, and logical operations), we opt not to propagate bounds information, but there is a choice: these instructions could also safely propagate bounds.

```
                         // Reg ← {value;  base;   bound}
1 set R1 ← 0x1000        // R1 ← {0x1000; 0;      0}
2 setbound R2 ← R1,4     // R2 ← {0x1000; 0x1000; 0x1004}
3 load R3 ← Mem[R2+2]    // read address 0x1002, check passes
4 load R3 ← Mem[R2+5]    // read address 0x1005, check fails
5 add R4 ← R2 + 1        // R4 ← {0x1001; 0x1000; 0x1004}
6 load R5 ← Mem[R4+2]    // read address 0x1003, check passes
7 load R5 ← Mem[R4+5]    // read address 0x1006, check fails
```

Figure 2. Code demonstrating implicit bounds checks and bounds propagation.

```
(A) add R1 ← R2 + imm  (output: R1, inputs: R2, imm)
      R1.value ← R2.value + imm  // do the addition
      R1.base ← R2.base          // copy R2's base
      R1.bound ← R2.bound        // copy R2's bound

(B) add R1 ← R2 + R3   (output: R1, inputs: R2, R3)
      R1.value ← R2.value + R3.value
      R1.base ← if (R2.bound != 0) R2.base else R3.base
      R1.bound ← if (R2.bound != 0) R2.bound else R3.bound

(C) load R1 ← Memory[R2]  (output: R1, inputs: R2)
      if (R2.base == 0 and R2.bound == 0) // nonpointer check
        raise non-pointer exception
      // do the bounds check
      if (R2.value < R2.base or R2.value ≥ R2.bound)
        raise bounds check exception
      else
        R1.value ← Mem[R2.value].value   // load value
        R1.base ← Mem[R2.value].base     // load base
        R1.bound ← Mem[R2.value].bound   // load bound
      endif

(D) store Memory[R2] ← R1   (inputs: R1, R2)
      if (R2.base == 0 and R2.bound == 0) // nonpointer check
        raise non-pointer exception
      // do the bounds check
      if (R2.value < R2.base or R2.value ≥ R2.bound)
        raise bounds check exception
      else
        Mem[R2.value].value ← R1.value   // store value
        Mem[R2.value].base ← R1.base     // store base
        Mem[R2.value].bound ← R1.bound   // store bound
      endif
```

Figure 3. Bounds propagation through `add` instructions `(A)` and `(B)` and bounds checking and propagation through a `load` `(C)` and `store` `(D)` instructions.

[1] The hardware also provides `readbound` and `readbase` instructions to allow the software to explicitly extract the pointer metadata.

Propagating bounds information to and from memory. Just providing an in-register representation for bounded pointers could reduce the runtime overhead of performing bounds checking, but it does not address the memory layout issues with fat pointers or reduce the overhead of storing and loading pointer values. To address these problems, the hardware also propagates bounds information to and from memory.

Just as all HardBound registers conceptually have extra base and bounds metadata, every value in memory also conceptually has a base and bound word associated with it (`Mem[addr].value`, `Mem[addr].base`, `Mem[addr].bound`). For example, the behavior of simple loads and stores is shown in Figure 3(C) and (D). Memory operations with more sophisticated addressing modes (register+immediate, register+register) are defined analogously.

A naive implementation of this conceptual model would triple the memory footprint, cache accesses, cache miss rates, and TLB miss rates. Section 4 describes our hardware encoding that more efficiently encodes the common cases of (1) non-pointer data and (2) a pointer in which the difference between base and bound is small, dramatically reducing the overheads from this worst-case scenario.

HardBound instructions are non-privileged. The proposed hardware is used solely to improve the efficiency and compatibility of fat pointers. Just as software-only fat pointers are manipulated and checked by user-mode instructions, our proposed hardware support also operates in user mode. The hardware *does not* provide protection in the sense of virtual memory, unforgeable capabilities [13, 37, 53] or fine-grained memory protection [60]. Although these privileged protection schemes are valuable, such support is not required to allow a compiler to generate a program binary that prevents all spatial memory violations.

3.2 Compiler and Runtime Support

HardBound's primitives are intended to provide spatial safety with minimally invasive changes to the compiler and runtime and without needing help from the programmer (in the form of source-code modifications). This section describes a variety of techniques for using hardware bounded pointers. Applying all of these techniques (as our prototype compiler does) achieves spatial safety guarantees as strong as those of CCured using only localized changes to the compiler.

Protecting heap-allocated objects. Heap-allocated objects are bounded by instrumenting `malloc()` and related runtime-library functions to appropriately set the base and bounds on pointers they return. For example, consider a program that `malloc()`s an array of characters. Once allocated, any array-indexed dereferences from that pointer are checked by the hardware. Likewise, any pointers created by performing arithmetic on this pointer (*e.g.*, repeatedly incrementing this pointer while iterating down the array) will also be checked when dereferenced. If this pointer is passed as a parameter or written to memory, the bounds information propagates without any further software intervention. Without additional compiler changes, using just this library instrumentation provides spatial safety of heap objects on a per-object granularity, even for compiled legacy code.[2]

Protecting local and global variables. The compiler performs a simple analysis to identify any pointers the program creates to local (stack-allocated) or global data structures, including stack-allocated arrays, global arrays, and any local or global variable passed by reference or whose address is taken (*e.g.*, `i` in `int i; int *j = &i;`). Once identified, the compiler adds an appropriate `setbound` instruction at the time the pointer is created (*e.g.*, `int *j = setbound(&i, 4);`). The sizes of all global objects and all stack-allocated objects are statically known, so the compiler already has the proper bounds information needed for the `setbound` instruction.

Once set with the proper bounds, the pointer to a global or stack-allocated object acts as a heap-allocated object from a bounds checking point of view. Because `setbound` is just an instruction that manipulates register values, it can be hoisted out of loops and otherwise optimized by normal compiler transformations.

Protecting sub-objects. A second case handled by the compiler is that of narrowing the bounds when creating a pointer to a sub-object within a larger object. For example, C allows statically-sized arrays to be embedded in a `struct`. If the program creates a pointer to such an array, the compiler refines the bounds to include just the extent of the array.[3] For example:

```
1   struct {char str[5]; int x;} node;
2   char *ptr = node.str;
3   strcpy(ptr, "overflow"); // overwrites node.x
```

Without intervention, the function call in line 3 would overwrite `node.x`. Although the size of the internal `str` array is known statically in the above code, the code for `strcpy()` has no way to check the bounds without help. To prevent this violation of spatial safety, the compiler rewrites line 2's access to `node.str` to refer to a pointer that has its bounds set appropriately using `setbound`:

```
2'  char *ptr = setbound(node.str, 5);
```

By setting the bounds on `ptr`, HardBound ensures that the spatial violation will be detected within `strcpy()`.

Programmer-specified sub-bounding. Extracting a single element from an array in C is an ambiguous operation with respect to bounds propagation. Given the statement `int* p = &q[3]`, without whole program analysis, the compiler cannot determine whether to propagate the bounds of the entire array `q` or to shrink the bounds to the sub-bounds of the single array element `q[3]`. Although it is always correct to maintain the bounds of the entire array, sub-bounding—if in line with the program's intent—both offers finer-grained protection and can reduce HardBound's overhead. Our compiler acts conservatively by not shrinking bounds, but such sub-bound operations were inserted in one benchmark to reduce overhead (see Section 5.3).

Programmer-specified (un)checked pointers. Finally, it is possible that the programmer knows (or can compute) appropriate bounds information that the compiler and runtime libraries do not have available. Such is the case with custom memory allocators, device drivers, and other low-level system code. Sophisticated programmers can write such code that is still safe by calling the `setbound` instruction directly. For example, a custom memory allocator that hands out chunks of a large array would follow the strategy of refining the bounds for the pointers to chunks it hands out. It is possible to construct a completely unsafe pointer that passes all bounds checks by setting base to zero and bound to `MAXINT`. This ability plays the same escape-hatch role in HardBound as C#'s unmanaged code or Java's JNI. As with any unsafe code extensions to a safe language, such pointers must be used carefully to ensure safety, but making unsafe operations explicit makes it easier to identify potential problems in the code.

[2] This `malloc()`-only HardBound mode checks memory accesses only when bounds information is present; no checking is performed on the non-heap references (*i.e.*, those memory accesses without bounds information).

[3] To handle the somewhat common idiom of dynamic over-allocation of `struct`s with zero-sized arrays as the last element, if the static size of the array is zero, the compiler generates a new pointer with base at the start of the array and bound that extends to the size of the `malloc`ed region.

4. Hardware Implementation

Although the model of hardware-supported bounded pointers presented in the last section is conceptually simple, a straightforward implementation would result in significant overheads. This section describes (1) placing the metadata into the virtual memory space, (2) adding a tag metadata space that identifies each word as a pointer/non-pointer, and (3) a compressed encoding of bounded pointers that dramatically reduces the overheads versus a naive implementation.

4.1 Placing Metadata in Virtual Memory

The base and bound metadata for memory words are placed in the virtual memory space, paralleling the normal data space, but offset by a constant amount. To improve spatial locality (and reduce fragmentation) of the base/bound region, these values are interleaved, allowing both the base and bound words to be read/written using a single double-word load/store operation. The function for calculating the address of the base and bound from a regular address is:

```
base(addr)  = SHADOW_SPACE_BASE + (addr * 2)
bound(addr) = SHADOW_SPACE_BASE + (addr * 2) + 1
```

These are normal virtual addresses, and accessing them follows the usual address translation, page allocation, and page swapping mechanisms in the operating system. `SHADOW_SPACE_BASE` is set when the address space is created, and it is stored in a special-purpose hardware register.

4.2 Encoding Pointer/Non-Pointer Efficiently

In a straightforward implementation, every load or store instruction would implicitly access the base and bound (as well as the data). Because the base and bounds shadow space is twice the size of the space of values, this would dramatically increase the cache and memory working set of the program. However, most of the values in C programs are non-pointers (represented by base and bound set to zero), and no base/bound accesses should be required for them.

To reduce the overhead of non-pointers, we introduce another metadata space—the *tag metadata space*. It contains one bit per word in memory to encode whether the word is a pointer or not. Before reading the base/bound metadata, the processor first checks the tag. If the word is not a pointer, the processor elides the access to the base/bound region. Whenever the processor writes a non-pointer word to memory, it clears the corresponding tag metadata bit. Whenever it writes a pointer to memory, it sets the tag metadata bit (in addition to writing the base/bound metadata). The tag metadata space uses only one bit per word, which adds only a few percent to the memory footprint (1 bit per 32-bit word is 3%).

Because this tag metadata is needed by every memory operation, we add a *tag metadata cache* as shown in Figure 4. The processor accesses this cache in parallel with the L1 cache. If the tag metadata indicates the location is a pointer, the processor then initiates a cache access to obtain the base/bound metadata. The tag cache is just a normal cache (same block size, dirty bits, coherence permissions), except that it caches blocks of metadata bits only. This cache is a peer with the primary data and instruction caches, and—just as for them—a miss in it will query the second-level cache before sending a request to memory (allowing caching of metadata in the second-level cache). Just as the instruction and data caches have dedicated address translation structures (*i.e.*, TLBs), the tag metadata cache also has its own TLB. As metadata tags are much smaller than data, the tag cache can be much smaller than the primary data cache—a 2KB tag cache holds the tag state of 64KB of the program's data (which is the size of a typical L1 data cache). This tag metadata cache organization is similar to MemTracker's split cache configuration for tracking auxiliary state [59].

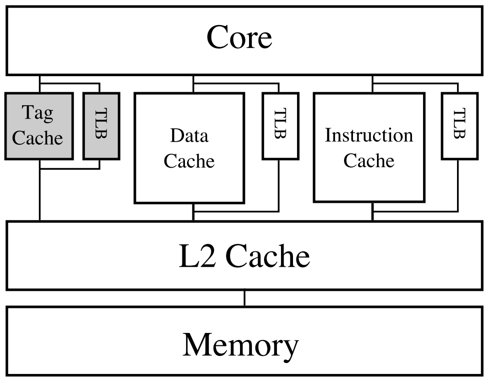

Figure 4. Placement of the tag metadata cache (shaded).

4.3 Compressing Bounded Pointers

Many pointers in C programs point to `structs` or small arrays. CCured's success in inferring `SAFE` pointers indicates that often the value and base component of a pointer are identical. Furthermore, most C `structs` are small so the difference between the pointer base and bound is also small. These observations suggest a simple mechanism for compressing the metadata: use just a few bits to encode the common case of pointers to small objects, but retain the full base/bound encoding option as a backup. For example, if a pointer's value and base are the same and the object size is less than 2^n, the base/bound metadata can be encoded in just n additional bits. We explore both *external compressed encodings* (additional bits in the tag space) and *internal compressed encodings* (opportunistically stealing redundant bits from within the pointer itself). By eliminating many accesses to the full base/bound metadata, these compressed encodings reduce cache accesses, cache capacity pressure, and physical pages allocated for metadata.

External compressed pointer encoding. By expanding the tag metadata space from one bit to four bits per word, the hardware can encode 2^4 tags to indicate whether a word is non-pointer data, one of 14 compressed bound sizes, or whether it is a non-compressed pointer. We found that most object sizes were multiples of four bytes, so we use the 14 patterns to compress pointers to the beginning of objects (*i.e.*, `base = ptr`) whose size ranges from 4 to 56 bytes (*i.e.*, `bound = tag*4`). The tag is set to "non-compressed" if (1) the size is not a multiple of four bytes, (2) the object is larger than 56 bytes, or (3) the pointer does not point to the beginning of the object. Non-compressed pointers are handled as described previously: the hardware accesses the full metadata.

Internal compressed pointer encoding. To avoid the runtime overheads of a larger tag metadata space, bits within the pointer itself can be transparently hijacked to encode metadata information. First, we steal one bit from the virtual address space to specify whether a pointer is compressed or not. By selecting this bit to correspond with the virtual memory region of the metadata shadow space, the total virtual memory space available is not further reduced. Note that this scheme still needs the 1-bit tag metadata that determines whether a location is a pointer, otherwise an integer that is cast to a pointer could masquerade as a bounded pointer, weakening the type safety guarantees discussed in Section 6.1.

Internal pointer compression targets pointers whose upper n bits are all ones or all zeros. For such pointers, it repurposes $n-1$ of these upper bits as metadata and the remaining bit is used to reconstruct the other $n-1$ bits of the pointer value during decompression. On 32-bit processors, using four internal bits would allow compression similar to the 4-bit external encoding above. However, it has the additional restriction that pointers to objects beyond the highest or lowest 128MBs of the virtual memory space are not eligible for compression. Pointers to objects in those regions still work correctly; they are just non-compressible.

For a 64-bit virtual address space, even for n as large as 14, only objects beyond the first *petabyte* (2^{50}) of the virtual address would be ineligible for compression. These additional bits enable the encoding of pointers to larger objects and more flexible pointer encodings (*e.g.*, allowing the base and pointer to be different).

4.4 Processor Core Operation

The processor core has four duties: (1) storing and propagating the metadata information in registers, (2) performing the bounds checks on memory operations, (3) loading and storing in-memory metadata, and (4) decompressing and opportunistically compressing pointers. A straightforward implementation for representing the register metadata is to add a double-word base/bound shadow register file (or use register sidecars [60]) and some datapath elements. This circuity operates in parallel with the main core pipeline, and its calculations are not on the critical path. A `setbound` instruction writes the base and bound into the shadow register file. Non-memory instructions copy the base/bound metadata from the input's shadow register to the output's shadow register. When dereferencing a pointer, the processor calculates the effective address and compares it to the base and bound; this bounds checking is done in parallel with the data cache lookup using a dedicated ALU.

The processor is also responsible for loading and storing pointer metadata. It uses a dedicated cache to access the tag metadata in parallel with the data cache for all memory operations. In contrast, the base/bound metadata and program data share the primary data cache. Base/bound metadata lookups—needed only when loading a non-compressed pointer—are performed sequentially, sharing the same cache port as the main pipeline. When writing a pointer to memory, the processor first determines whether the pointer is compressible. For all stores, the processor then writes the data cache (with the actual data value of the store) and the tag metadata cache. When storing an incompressible pointer, the processor performs an additional data cache write to update the full base/bound metadata.

Manipulation of compressed pointers presents a design choice. In our baseline implementation, the hardware expands compressed pointers whenever they are loaded by writing the expanded base and bound into the shadow registers. Alternatively, the processor can use the compressed pointer representation internally by adding tag metadata sidecars to the primary registers. In the latter approach, the base/bound shadow register file is accessed only when manipulating non-compressed pointers. When dereferencing compressed pointers, a narrow adder checks that the address is in bounds. Whenever a pointer's value changes (*e.g.*, due to a pointer increment), if the resulting pointer is no longer compressible, the hardware expands the pointer. Finally, for any instruction that uses a pointer value (*e.g.*, comparing the equality of two pointers), the hardware uses the actual pointer values (not the compressed ones) in the computation. By following these invariants, the compressed encodings remain invisible to programs running on the processor.

4.5 Other Implementation Considerations

Forward compatibility. If `setbound` is given an instruction encoding that is currently a no-op instruction, newly annotated programs can be distributed widely (to both those users with the additional hardware support and those without). Running a modified binary on a current processor will execute just as a normal C program. However, once the user upgrades to a new machine with the appropriate hardware support, these same binaries will begin providing spatially protected execution.

OS support. The only operating system change required is saving and restoring the additional architected state (base/bound shadow registers and a few control registers) on context switches. Such a change is required any time new architected registers are added, for example, when Intel added registers for their MMX/SSE/SSE2 extensions. Because the base/bound metadata and tags are placed in the virtual memory space, no special paging support is needed.

Atomic pointer operations. As with software-only fat pointer approaches, if operations on bounded pointers occur non-atomically, interleaved execution and race conditions (either unintentional races or intentional races used in lock-free concurrent data structures) can cause memory safety violations [22]. To provide thread-safe execution, HardBound performs the pointer access and the metadata accesses as a single atomic operation. This operation is essentially a bounded memory transaction [25, 33], and it can be implemented using any of the proposed hardware transactional memory techniques. Hardware support for bounded transactions could be used to provide atomicity for software-based fat pointer operations as well.

5. Experiments

We evaluate HardBound by (1) testing its ability to detect memory safety violations and (2) assessing its runtime overheads. For these tests, we use programs compiled with our prototype compiler that inserts `setbound` instructions to enforce complete spatial safety.

5.1 Experimental Methods

We use the Simics full-system simulator [38] to simulate an in-order 32-bit x86 processor. We simulate all user-mode code (including DLLs) and kernel-mode instructions, but HardBound is disabled while executing kernel code. We use PTLSim [63] to decode x86 instructions into micro-operations (μops). The simulated processor executes at most one micro-operation per cycle. Any load or store of an uncompressed bounded pointer creates an additional micro-operation to access the bounds metadata. The simulated memory hierarchy models a 32KB 4-way SA first-level cache with a 12-cycle miss penalty, a 4MB 4-way SA L2 cache with 200-cycle miss penalty, and 4-way SA 256-entry TLBs with 4KB pages with a 12-cycle miss penalty. The tag metadata cache is 2KB 4-way SA when HardBound uses a 1-bit encoding; it is 8KB 4-way SA when using a 4-bit external compressed encoding. All caches have 32-byte blocks.

The prototype compiler inserts `setbound` instructions using a set of CIL source-to-source transformations [41]. The resulting C code is then passed to GCC 4.2 with -O3 optimizations.

We compare against CCured as a representative of a highly-optimized software-only scheme. CCured incurs runtime overheads both for providing spatial safety (using fat pointers) and temporal safety (using a conservative garbage collector and selective heapification of stack-allocated variables). As HardBound focuses only on spatial safety, we reduce CCured's overhead by disabling these temporal safety features to provide a fair performance comparison.

We chose the Olden benchmarks [48] for our performance evaluation because they are pointer intensive and have been used to evaluate important prior works (*e.g.*, [2, 14, 40]), allowing comparisons. Furthermore, as C programs do require some changes to work correctly and efficiently with CCured, we obtained modified sources for the Olden benchmarks directly from the CCured group.

5.2 Functional Correctness Experiments

We verified the functional correctness of our scheme by testing it against a suite of 291 spatial memory violations [31]. The suite contains a wide range of spatial violation tests, including various combinations of: reads and writes; upper and lower bounds; stack, heap, and global data segments; and various addressing schemes and aliasing situations. Each test case has two versions: one with

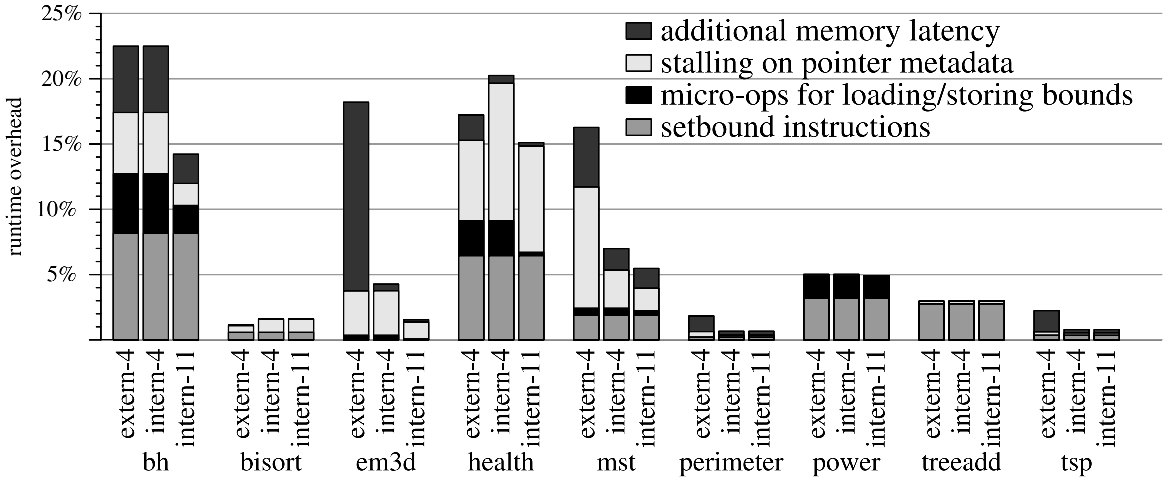

Figure 5. Benchmark runtime results

the violation and one without, to allow testing for false positives. We ran all but five of the test pairs—the omitted tests are incompatible with our simulation environment because they use pthreads, fork, Unix shared memory segments, or timers.

Of the remaining 286 test pairs, HardBound detects all the violations and generates no false positives. We also successfully ran 77 additional programs as part of our own testing infrastructure. None of these correctness results should be surprising—other fat pointer schemes already provide complete spatial safety. However, these results significantly increase our confidence that our performance simulations capture pointer dereferences and propagation correctly.

5.3 Source Code Modifications

None of the simulated programs required any source code modifications (including the original Olden benchmarks and the CCured versions) for correct operation. This finding is consistent with other approaches that avoid changing memory layout (*e.g.*, [14, 28]).

However, during the course of evaluating HardBound, we made two performance-related changes. The first change addressed an artificial limitation of our prototype compiler—it creates bounded pointers even for constant-index array references to stack-allocated arrays. To mitigate the impact of this limitation, we restructured the code for `bh` by manually inlining two functions, avoiding redundant calls to `setbound` in an inner loop.

Second, in three places `mst` uses a pointer into the middle of an array as a pointer that references a particular array element exclusively, instead of as a pointer to the entire array. Because our compiler acts conservatively in this inherently ambiguous situation (as discussed in Section 3.2), we inserted `setbound` instructions to tighten bounds in these three cases. This better expresses the intended constraints of the program and reduces overheads by avoiding the propagation of difficult-to-compress pointers.

5.4 Runtime Overheads

We first report the runtime overheads of instrumented binaries relative to unmodified binaries, both compiled with GCC (we compare against CCured in the next subsection). Figure 5 shows the relative runtime of three different pointer encoding schemes. The four segments in each bar represent the runtime contributions of: (1) extra `setbound` instructions inserted by the compiler, (2) extra micro-operations inserted for writing/reading the metadata of uncompressed pointers to/from the memory hierarchy, (3) cache misses on metadata (both compressed and uncompressed), and (4) additional cache misses caused by pollution from the metadata.

Pointer encoding impact. In Figure 5, the leftmost bar in each group uses a 4-bit external compressed encoding that can compress pointers to small objects ($\leq$ 56 bytes, size divisible by 4) where the pointer is equal to the base. For this, our simplest encoding, the average slowdown is only 9%, though several benchmarks incur significant runtime overheads (`bh`, `em3d`, `health`, `mst`).

The second bar in each group uses a 4-bit internal pointer scheme capable of encoding the same set of small-object pointers. The tag metadata shadow space is shrunk from 4-bits to 1-bit (and the tag metadata cache is reduced accordingly from 8KB to 2KB). The primary benefit of this encoding is that it reduces the size of the tag metadata cache. As a secondary benefit, it lowers the average overhead to 7%, primarily because it reduces the amount of pollution caused by tag metadata in the second-level cache.

The third bar in each group shows the runtime overhead of an 11-bit internal encoding in which both pointer equals base and (base - bound) $\leq 4 \times 2^{11}$ bytes, as discussed in Section 4. This 11-bit encoding would be suitable for a system with a 64-bit virtual address space. By reducing the number of incompressible pointers, this scheme trims the maximum runtime overhead to 15% and the average to only 5%.

Bounded pointer μop impact. The performance impacts discussed above include the cost of storing and loading uncompressed bounded pointers to and from memory (the dark bar second from the bottom). As these accesses share the same cache ports, they introduce runtime overheads. Fortunately, loading and storing of uncompressed pointers is rare for all three encoding schemes, limiting the μop overheads to typically only a few percent.

However, these results assume that a bounds check (for either a compressed or an uncompressed pointer) is done in parallel with the dereference and thus does not add additional cost. A more modest implementation might perform bounds checking of uncompressed pointers by using shared ALUs and register file ports. To examine the performance impact of such a design choice, we ran a simulation in which each bounds check of an uncompressed pointer inserts an additional μop into the processor (results not shown). The average overhead increased by approximately 3% for all three encodings, while the maximum was a 10% increase for `tsp`.

Memory usage overheads. To assess HardBound's impact on memory usage, we measured the number of additional distinct pages touched, compared to the baseline C versions. Figure 6 presents these results, using 4KB pages and excluding the effects of kernel code. Programs running under HardBound touch additional

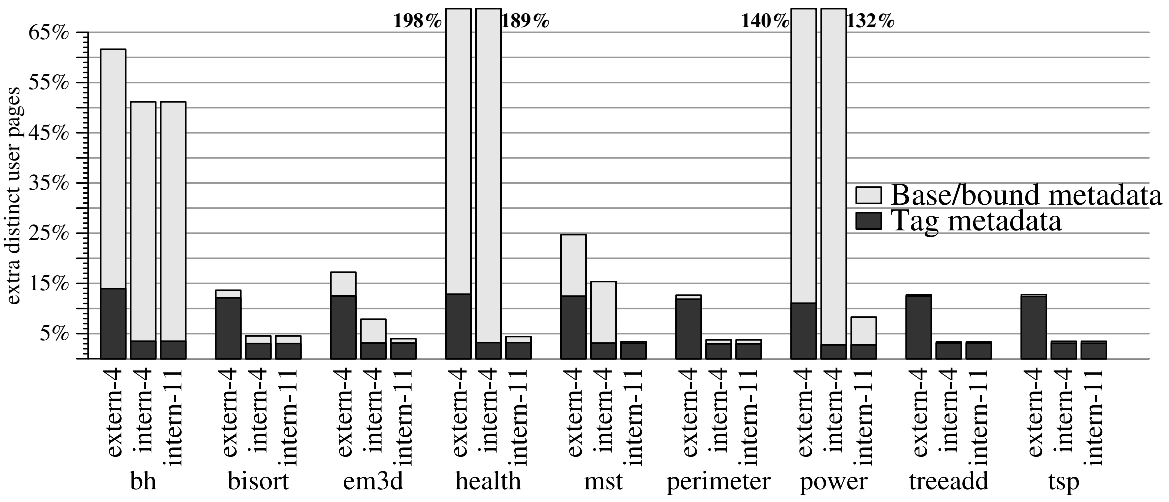

Figure 6. Benchmark memory overhead (normalized 4KB pages touched)

	JK/RL/DA	CCured						HardBound		
						Simulator		External	Internal	
Benchmark	**Published [14]**	**Published [40]**	**Pentium4**	**Core 2**	**Opteron**	**Uops**	**Runtime**	**4bit**	**4bit**	**11bit**
bh	1.00	**1.44**	**1.33**	1.18	**1.29**	**1.74**	**1.72**	**1.22**	**1.22**	1.14
bisort	1.00	1.09	1.09	1.07	1.09	**1.22**	1.20	1.01	1.02	1.02
em3d	**1.68**	**1.45**	**1.51**	**1.39**	**1.36**	**1.64**	**1.31**	1.18	1.04	1.02
health	**1.44**	1.07	0.99	1.01	1.01	**1.23**	1.11	1.17	1.20	1.15
mst	**1.26**	**1.87**	1.12	1.05	1.09	**1.39**	1.06	1.16	1.07	1.05
perimeter	0.99	1.10	**1.22**	**1.25**	**1.32**	**1.58**	**1.51**	1.02	1.01	1.01
power	1.00	**1.29**	**1.21**	1.02	1.10	**1.80**	**1.79**	1.05	1.05	1.05
treeadd	0.98	1.15	1.19	1.18	1.03	1.16	1.09	1.03	1.03	1.03
tsp	1.03	1.06	0.96	1.00	1.00	1.09	1.07	1.02	1.01	1.01
Average	*1.13*	***1.26***	*1.17*	*1.12*	*1.14*	***1.40***	***1.29***	*1.09*	*1.07*	*1.05*

Figure 7. Runtime overhead comparison of JK/RL/DA, CCured, and HardBound. Runtime overheads of over 20% are in bold. Data for columns two and three are from published papers. Data for columns four, five, and six were collected on a 3.2 GHz Pentium 4, a 2.66 GHz Core 2 Duo, and a 1.8 GHz Opteron. Data for columns six and seven are simulation results for micro-ops and runtime overhead for CCured.

memory for two reasons: (1) tag metadata and (2) base/bound metadata. Programs with a large number of incompressible pointers (*e.g.* `health` and `power` when run with a 4-bit encoding) touch a significant number of additional pages due to base/bound lookups. On average, the 4-bit external encoding touches 55% more pages than the baseline. As expected, the 4-bit internal encoding reduces the overhead of accessing the tag metadata, but does not affect the base/bound overhead as this scheme fails to compress the same pointers the external scheme does. The 11-bit internal encoding allows many more pointers to be compressed, attacking the base/bound overhead and reducing the average number of additional pages touched to just 10%.

5.5 Comparative Evaluation

Figure 7 compares the runtime performance overheads of two state-of-the-art software-only approaches with HardBound. The first two columns report the runtime overheads of JK/RL/DA [14] and CCured [40] as reported in the respective publications. These two proposals are representative of object-based (Section 2.2) and fat-pointer (Section 2.3) software-only approaches. The overheads for JK/RL/DA are normalized to a baseline that includes their synergistic automatic pool allocation optimization. The published runtime overheads for both JK/RL/DA and CCured are small on average (13% and 26%), but some benchmarks have slowdowns over 20% (marked in bold).

The CCured published data uses a different compiler version and includes garbage collection and other overheads related to temporal safety, so we also ran our own experiments with CCured without these overheads, and using the same compiler infrastructure as in our other experiments. The third through fifth columns report that the average overhead of CCured on three x86 machines is around 15%. As before, in several cases the runtime overheads of CCured on real machines exceeds 20% (marked in bold).

The sixth and seventh columns report results for these CCured binaries under simulation. Comparing the μop count in the sixth column to the previous columns indicates that CCured introduces a large number of instructions in some cases, but the ILP of these modern processors hides much, but not all, of the overhead. As shown in column seven, our simulated in-order processor lacks ILP to hide the cost of bounds checks, resulting in overheads higher than on the actual hardware (but significantly smaller than the number of added micro-operations due to time in the memory system).

The three right-most columns show the relative runtimes for HardBound with the three different pointer encoding schemes (replicated from Figure 5). As reported above, the average runtimes for these schemes are all less than 10%. Note that unlike CCured and JK/RL/DA, for which some benchmarks have large slowdowns, the largest runtime overhead for the 11-bit internal encoding is only 15%.

6. Handling Casts and Temporal Safety

Beyond spatial memory safety, which is HardBound's focus, two other significant sources of errors in C programs are type safety violations (via unsafe casts) and temporal memory safety violations (due to dangling pointers, uninitialized reads, and misuse of free). Although HardBound's fat pointers are tailored to the problem of spatial safety, they provide sufficient type safety to prevent spatial violations while allowing legitimate programs to run; they have some synergies with mechanisms for providing temporal safety too.

6.1 Type Safety and Casts

From HardBound's point of view, C cast operations are no-ops. Consequently, HardBound's metadata propagation is unaffected by casts (and union accesses), and, as a result, the types declared in the C program are not taken literally. Because the hardware's treatment of metadata distinguishes between non-pointer and pointer data dynamically, one can think of HardBound as providing (coarse-grained) runtime-type information. This means that it is not possible to create a useful pointer in HardBound without using the setbound instruction—casting an int constant to an int* results in a non-pointer that will fail all bounds checks if dereferenced, which is the desired behavior for preventing spatial memory safety violations. HardBound distinguishes data pointers from code pointers (by setting base and bound to MAXINT) to prevent forging of arbitrary function pointers, even in the presence of unsafe casts.

Casting a value from a pointer type to another type propagates the bounds information without change. This implies that upcasting (from a larger struct to a smaller, structurally compatible one) is fine. Downcasts may result in bounds violation errors that are caught only when the offending pointer is dereferenced, but if the code is correct no errors should result. Similarly, it is possible for correct code to upcast a pointer to a void*, and then downcast it back to a non-void*.

For example, consider the following code fragment:

```
1  int x = 17;
2  char y = (char) x;      // legal cast (just a mov)
3  char *z = (char *)&x;  // compiler inserts bounds on z
4  int a = (int)z;         // a inherits z's bounds
5  (*(int *)a) = 42;       // legal update (x is now 42)
6  int *w = (int *)0x1000;// no bounds info for w
7  *w = 42;                // illegal write detected
```

The cast on line 2 could be considered unsafe, because it converts an int to a char, but the hardware will permit this without any problem. Taking the address of the variable x (line 3) causes the compiler to add bounds information on the pointer value stored in z. It is possible to cast such a pointer to an int and back again and still write through the resulting pointer (lines 4 and 5). If the program manufactures a pointer out of a constant, as in line 6, then any read or write through that pointer will fail (line 7). If absolutely necessary, a programmer can still create a pointer from an integer by explicitly inserting a setbound instruction.

This default design requires no additional compiler support, and it provides just enough dynamic type checking to guarantee spatial memory safety. In essence, HardBound's spatial and type guarantees are the same as giving all pointers the semantics of CCured's WILD pointers (but without the runtime overhead of WILD pointers). To provide stronger type safety, additional compiler and runtime support could be used to create type information and dynamically check potentially unsafe casts. CCured [40] uses run-time type information to handle casts; SAFECode [16] uses static analysis to partition the heap based on type information.

6.2 Temporal Errors

C also suffers from temporal memory safety problems, most notably uninitialized memory reads and dangling pointers. The initialization problem can be remedied by forcing malloc() to zero-out memory before reallocation; similarly, the compiler can insert initialization code for all local variables and arrays.

Handling dangling pointers is more difficult. The approach used by CCured is to employ a conservative garbage collector [4]. Under such a system, the free() operation does no work (avoiding problems with double-frees), and no heap pointers can dangle, because any object reachable by pointer traversal is ensured not to be deallocated. To prevent dangling pointers to stack objects, CCured selectively heapifies stack objects that escape the function. Applying garbage collection in a HardBound system would have the further advantage that HardBound's metadata precisely distinguishes pointers from non-pointers, opening up the possibility for non-conservative garbage collection of C.

Because garbage collection remains undesirable in many application domains, it is worth considering alternative approaches for temporal safety. Other proposals that address temporal errors, either probabilistically [3, 44, 50] or exactly [2, 15, 17, 21, 23, 26, 28, 45], are compatible with our hardware.

Finally, Purify [24] and Valgrind's MemCheck [43] keep track of the allocated/unallocated status of each word in memory to catch many (but not all) dangling pointer dereferences. Recent proposals [47, 59, 65] have explored accelerating such tracking in hardware. As HardBound already tracks a bit of metadata per word in memory (pointer vs non-pointer), adding such additional tracking to HardBound would be a natural extension.

7. Additional Related Work

In addition to works described earlier, there have been many hardware approaches proposed for handling security issues in C-based programs.

Taint checking and intrusion detection. Some proposals seek 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 [10], LIFT [46], RIFLE [58], the work by Suh *et al.* [55], and Raksha [12]. Other techniques seek to detect anomalous behavior [20, 30, 66] or to combine tainting and bounds checking [6]. In contrast to approaches that provide complete spatial 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 a 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.

Cryptographic and tamper resistant hardware. There has also been much recent work on hardware support for cryptographically sealed code [18], encrypted memory [36, 51, 54], secure processors [35, 52, 56], and tamper resistant hardware [36, 54]. Although these efforts are largely orthogonal to the spatial safety support proposed here, 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 errors. In fact, the spatial errors we prevent could otherwise allow attackers to bypass the tamper resistance by taking over the device's software.

8. Conclusions

This paper introduces HardBound, a cooperative hardware/software approach for enforcing spatial safety of C programs. Its goal is to eliminate the spatial memory errors that are the source of so many bugs and security vulnerabilities. HardBound provides a hardware bounded pointer datatype and the processor automatically checks and propagates bounds. HardBound's key advantages over software-only approaches are lower runtime overheads, better source code and binary compatibility, and a simpler compiler infrastructure, all of which are important for widespread adoption.

We implemented both a prototype compiler and hardware simulator and studied several metadata compression schemes. Functionally, HardBound accurately detected and prevented all spatial memory violations in hundreds of test cases with no false positives. Performance-wise, our experiments suggest that HardBound has low overhead (less than 10% on average), which is lower than prior software-only techniques.

Looking forward, HardBound can be viewed as complementary to the optimization techniques developed to accelerate software-only approaches. For example, CCured could use HardBound pointers for representing `SEQ` and `WILD` pointers (but not `SAFE` pointers), further reducing overheads versus either technique alone. Similarly, if the compiler can statically prove that bounds checking is not necessary, it can unbound the pointer to reduce HardBound's checking overheads. Finally, HardBound could be employed to reduce the runtime costs of checking array bounds in already-safe languages such as Java or C#.

Acknowledgments

The authors thank Vikram Adve, Emery Berger, Mike Hicks, E Lewis, Andrew Myers, Santosh Nagarakatte, and Amir Roth for comments on this work. We thank the CCured group for making their software available, and Westley Weimer and Matt Harren in particular for helping us with the CCured benchmarks. This work is supported in part by donations from Intel Corporation and NSF awards CCF-0541292, CNS-0524059, and CCF-0644197.

References

[1] 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.

[2] 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.

[3] 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.

[4] H.-J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software — Practice & Experience*, 18(9):807–820, Sept. 1988.

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

[6] 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.

[7] 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.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Conference*, 2003.

[9] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, Jan. 1998.

[10] 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.

[11] 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.

[12] 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.

[13] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, 1966.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] D. Grossman. Type-Safe Multithreading in Cyclone. In *Proceedings of the SIGPLAN Workshop on Types in Languages Design and Implementation*, Jan. 2003.

[23] 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.

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

[25] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] K. Kratkiewicz and R. Lippmann. Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[32] L. Lam and T. Chiueh. Checking Array Bound Violation Using Segmentation Hardware. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2005.

[33] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

[34] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the International Conference on Security in Pervasive Computing*, Mar. 2003.

[35] 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.

[36] 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.

[37] T. A. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4):409–445, 1976.

[38] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[39] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee. A Processor Architecture Defense against Buffer Overflow Attacks. In *Proceedings of the IEEE International Conference on Information Technology: Research and Education*, Aug. 2003.

[40] 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.

[41] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *11th International Conference on Compiler Construction*, 2002.

[42] 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.

[43] 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.

[44] 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.

[45] H. Patil and C. N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *Second International Workshop on Automated Debugging*, May 1997.

[46] 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.

[47] 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.

[48] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, 1995.

[49] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, Feb 2004.

[50] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: A Helper-Thread Approach to Programmable, Automatic, and Low-Overhead Memory Bug Detection. *IBM Journal of Research and Development*, 50(2/3):261–275, 2006.

[51] 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.

[52] 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.

[53] F. G. Soltis. *Inside the AS/400*. Duke Press, 2nd edition, 1997.

[54] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th International Conference on Supercomputing*, June 2003.

[55] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[56] 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.

[57] N. Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[58] 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.

[59] 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.

[60] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[61] 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.

[62] 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.

[63] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr 2007.

[64] 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.

[65] 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.

[66] 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.