SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic

Computer and Information Sciences Department, University of Pennsylvania

santoshn@cis.upenn.edu jianzhou@cis.upenn.edu milom@cis.upenn.edu stevez@cis.upenn.edu

Abstract

The serious bugs and security vulnerabilities facilitated by C/C++'s lack of bounds checking are well known, yet C and C++ remain in widespread use. Unfortunately, C's arbitrary pointer arithmetic, conflation of pointers and arrays, and programmer-visible memory layout make retrofitting C/C++ with spatial safety guarantees extremely 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 paper proposes SoftBound, a compile-time transformation for enforcing spatial safety of C. Inspired by HardBound, a previously proposed hardware-assisted approach, SoftBound similarly records base and bound information for every pointer as disjoint metadata. This decoupling enables SoftBound to provide spatial safety without requiring changes to C source code. Unlike HardBound, SoftBound is a software-only approach and performs metadata manipulation only when loading or storing pointer values. A formal proof shows that this is sufficient to provide spatial safety even in the presence of arbitrary casts. SoftBound's full checking mode provides complete spatial violation detection with 67% runtime overhead on average. To further reduce overheads, SoftBound has a store-only checking mode that successfully detects all the security vulnerabilities in a test suite at the cost of only 22% runtime overhead on average.

Categories and Subject Descriptors D.3.3.4 [*Programming Languages*]: Processors; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Performance, Security, Reliability

Keywords spatial memory safety, buffer overflows, C

1. Introduction

The serious bugs and security vulnerabilities facilitated by C/C++'s lack of bounds checking are well known. The lack of *spatial 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 [14, 38, 44]. Even though modern operating systems and compilers employ partial countermeasures (*e.g.*, guarding the return address on the stack, address space randomization, non-executable stack), vulnerabilities persist. For one example, in November 2008 Adobe released a security update that fixed several serious buffer overflows [2]. Attackers have reportedly exploited these buffer-overflow vulnerabilities by using banner ads on websites to redirect users to a malicious PDF document crafted to take complete control of the victim's machine [1]. 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 [39].

Safe languages, such as Java and C#, enforce memory safety and thus completely prevent this entire class of bugs and security vulnerabilities [14]. Such languages have thankfully become mainstream, however C and C++ are still widely used. C provides low-level control of memory layout, proximity to the underlying hardware, requires minimal runtime support, and is the gold standard for performance. Today's operating systems, virtual machine monitors, language runtimes, enterprise database management systems, embedded software, and web browsers are all generally written in C/C++. Furthermore, altogether such systems comprise millions of lines of C/C++ code, preventing the complete transition away from C/C++ anytime soon.

As a recognition to the importance of this problem, many proposals have pursued techniques for retrofitting C (or close variants) to provide complete or near-complete spatial memory safety [4, 5, 10, 11, 17, 28, 29, 32, 35, 37, 41, 47, 48].[1] Unfortunately, several aspects of C, such as its conflation of arrays and singleton pointers, unchecked array indexing, pointer arithmetic, pointers to the middle of objects, arbitrary casts, user-visible memory layout, and structures with internal arrays all interact to greatly increase the difficulty of retrofitting C with spatial memory safety guarantees. As a result, prior proposals suffer from one or more practical difficulties that may prevent wide adoption, such as: 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 proposals with the lowest performance overheads generally employ whole-program compiler analyses (*e.g.*, [4, 17, 35]) which complicates separate compilation and use of dynamically linked libraries. Section 2 provides additional background on these proposals.

Hardware-assisted techniques have been proposed for mitigating the runtime overheads and other limitations of these software-

[1] Although temporal safety violations are also a source of bugs (*i.e.*, dangling pointers) and vulnerabilities (*i.e.*, use-after-free vulnerabilities), SoftBound focuses exclusively on the spatial safety issues of C. Other previously proposed complementary techniques such as conservative garbage collection [9], reference-counted smart pointers, probabilistic approximations of an infinite-sized heap [6], temporal capabilities [5, 47], or region-based memory management [19, 23, 25] may be employed to detect or prevent temporal violations.

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06. . . $5.00

only schemes. One such proposal is HardBound [16], which describes extensive hardware support for bounded pointers, including automatically propagating pointer bounds information, efficiently checking every memory access, and transparently recording pointer bounds metadata in a hardware-managed shadow space. This hardware/software approach may provide low-overhead enforcement of spatial memory safety that is complete and highly compatible with existing C code, but such hardware support is unlikely to be available any time soon, if ever.

This paper describes SoftBound, a software-only approach inspired by HardBound's disjoint metadata scheme for highly compatible enforcement of spatial safety for C programs. SoftBound is a compile-time transformation that inserts code for runtime propagation and checking of pointer bounds. SoftBound enforces spatial safety using the pointer-based approach, which associate base and bound metadata with every pointer. Unlike prior pointer-based approaches that change pointer representations and thus object layouts [5, 28, 35], SoftBound records the base and bound metadata in a disjoint metadata facility that is accessed via explicit table lookups on loads and stores of pointer values only. SoftBound performs a simple intra-procedural transformation that instruments each function to propagate and check pointer metadata. Functions with pointer arguments or pointer return values are extended with additional parameters for base and bound metadata. This overall approach provides SoftBound with the following attributes:

- **Source compatibility.** SoftBound is highly compatible with existing C source code because its disjoint metadata (1) avoids any program-visible memory layout changes, (2) allows arbitrary casts by preventing the coercion of metadata that could otherwise occur with in-line metadata. Our experiments with 23 benchmark applications and two network daemons (272K lines of code in total) show that SoftBound can be successfully applied to unmodified C programs.
- **Completeness.** By default the SoftBound transformation guarantees spatial safety. In essence, SoftBound provides the same spatial safety guarantees as CCured, and Section 4 includes the sketch of a formal proof that SoftBound's core mechanisms enforce a well-formed memory property similar to that provided by CCured [35]. Our experiments show that SoftBound catches errors not caught by Valgrind's memcheck tool [42] or GCC's Mudflap [21].
- **Separate compilation.** SoftBound's simple intra-procedural analysis, disjoint metadata, and function cloning provide transparent support for separate compilation, allowing library code to be recompiled with SoftBound and dynamically linked. This extends checking into library code and can reduce the need for the library wrappers used by prior proposals.

SoftBound provides two modes of checking. For low-overhead debugging and security-critical software, SoftBound's full checking mode provides spatial safety at the cost of a 67% runtime overhead on average over a range of 23 benchmarks. In contrast to heavyweight instrumentation used selectively during the development process [21, 27, 36, 42], SoftBound's full checking overheads are low enough to use continuously throughout the software development process. For security-critical applications, such overheads are also likely acceptable.

For lower overhead protection against security vulnerabilities and the memory corruption caused by out-of-bounds writes, SoftBound provides a store-only checking mode. In this mode, SoftBound propagates all metadata, but inserts bounds checks only for memory writes. As observed previously [4, 48], store-only checking is sufficient to stop just about any security vulnerability, because attacks typically require performing at least one out-of-bounds write. Out-of-bound writes are particularly subversive bugs in that the memory corruption caused by such bugs often manifests much later and in a different part of the program code. Our experiments show store-only SoftBound reduces the average runtime overhead to just 22% while still preventing all the vulnerabilities in a security vulnerability suite. SoftBound's store-only checking overhead is low enough for many benchmarks (over half of the benchmarks evaluated have less than 15% runtime overhead) to be used in end-user production code.

2. Background

The problem of detecting and preventing spatial violations in the C programming language is a well researched topic. Many techniques were initially proposed primarily as debugging aids [5, 29, 37], but have now been improved immensely [15, 17, 28, 35, 41, 47]—nearly to the point of being ready for deployment in production systems.

In this section, we describe approaches most closely related to our scheme, comparing them with respect to completeness, performance and compatibility attributes. Because SoftBound is focused on detecting spatial violations, this section does not discuss approaches for preventing temporal safety violations (*i.e.*, dangling pointers). Additional related work is discussed in Section 7.

2.1 Object-Based Approaches

Object-based approaches [15, 17, 21, 29, 41] 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 require special care. Object-based schemes use special out-of-bounds proxy objects [17, 29] 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 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 5x or more [21, 29, 41]. Subsequent proposals have considerably mitigated this issue, reducing the overhead of the object-table by checking only strings [41] or using whole-program analysis to perform automatic pool allocation to partition the splay trees and eliminate lookup and checking for many scalar objects [15, 17].

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:

```
struct {
  char id[8];
  int account_balance;
} bank_account;
char* ptr = &(bank_account.id);
strcpy(ptr, "overflow...");
```

In the above example code, 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. The above example is admittedly contrived, and we have not encountered any such sub-object overflows in the benchmark code we have examined. However, 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 [17, 18], the most recent and most advanced 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.

2.2 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. As two distinct pointers can point to the same object and have different base and bound associated with them, 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 [5], CCured [35], Cyclone [28], MSCC [47], and others [16, 36, 37] use this pointer-based approach to provide spatial safety guarantees.

The pointer-based approach is attractive in that it can be used to enforce complete spatial safety. However, propagating and checking bounds for all pointers can result in significant runtime overheads. To reduce these overheads, CCured [35] 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 typecasts. *SEQ* pointers are fat pointers that allow only pointer arithmetic and array indexing and are not involved in arbitrary typecasts. *WILD* pointers allow arbitrary 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.

The most significant disadvantage of the pointer-based approach is that fat pointers change memory layout in programmer-visible ways. This introduces significant source code compatibility issues in that the source must be modified [35]. The modified memory layout also makes interfacing with library code challenging. To address this issue, attempts have been made to split the metadata from the pointer [35, 47]. These approaches partially mitigate some of the compatibility issues, but such techniques can increase overhead by introducing linked shadow structures that mirror entire existing data structures. Furthermore, they do not handle arbitrary casts (another compatibility issue) and MSCC's [47] optimized encoding loses the ability to detect sub-object overflows.

Approach	No src code change	Detects sub-object violations	Memory layout compat.	Arb. casts	Dyn. link lib
J&K [29]	Yes	No	Yes	Yes	Yes
SafeC [5]	Yes	Yes	No	Yes	No
CCured – *Safe/Seq* [35]	No	Yes	No	No	No
CCured – *Wild* [35]	Yes	Yes	No	Yes	No
MSCC [47]	Yes	Yes/No	Yes	No	Yes
SoftBound	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 1. Comparison of representative object-based (Jones & Kelley) and pointer-based approaches (SafeC, CCured, MSCC) in contrast to SoftBound with respect to attributes such as: source code modification, completeness with respect to detecting subfield access violations, memory layout compatibility, support for arbitrary casts, and support for dynamically linked libraries.

2.3 Comparison of Various Approaches

Object-based and pointer-based approaches have complementary strengths and weaknesses. Object-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 [15]. 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 [17, 35], it can complicate the use of precompiled and dynamically loaded libraries.

Table 1 summarizes the various object-based and pointer-based approaches in contrast with SoftBound. Object-based approaches such as Jones & Kelley [29] 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 [47] uses split metadata and runtime 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. In the next section, we describe the SoftBound approach, which satisfies all the attributes listed in the Table 1 by combining the disjoint metadata of object-based schemes with the sub-object overflow detection of pointer-based schemes.

3. The SoftBound Approach

SoftBound is a compile-time transformation for inserting runtime bounds checks to enforce spatial safety of C programs. SoftBound is highly compatible with existing C source code because its disjoint metadata representation avoids memory layout changes and allows arbitrary casts. SoftBound associates base and bound metadata with every pointer, but records that metadata in a disjoint metadata space that is accessed via explicit table lookups. This approach is conceptually a pointer-based approach, but SoftBound's disjoint metadata provides the memory layout compatibility of object-based approaches. This section describes SoftBound's key ideas. Section 4 formalizes SoftBound and sketches a proof of SoftBound's spatial memory safety guarantee. A full discussion of SoftBound's specific implementation choices and its handling of all of C's features is deferred to Section 5.

3.1 Pointer Checking and Metadata Propagation

The following description of SoftBound's transformation assumes the C code has been translated into a generic intermediate form that contains only simple operations, uses explicit indexing and memory access operations, and provides the abstraction of an unbounded number of non-memory intermediate values and temporaries that will ultimately be mapped to registers.

Pointer dereference check For every pointer value in the program's intermediate representation, the SoftBound transformation creates a corresponding *base* and *bound* intermediate value. Whenever a pointer is used to access memory (*i.e.*, dereferenced), SoftBound inserts code (highlighted in grey) for checking the bounds to detect spatial memory violations:

```
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
value = *ptr;     // original load
```

Where `check()` is defined as:

```
void check(ptr, base, bound, size) {
  if ((ptr < base) || (ptr+size > bound)) {
    abort();
  }
}
```

The 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, if a pointer to a single character is cast to be a pointer to an integer, dereferencing that pointer is a spatial violation. This dereference check is inserted for all pointer dereferences, but such a check is not required when accessing scalar local or global variables, or when spilling/restoring register values to/from the stack—we assume that the C compiler generates such code correctly.

Creating pointers New pointers in C are created in two ways: (1) explicit memory allocation (*i.e.* `malloc()`) and (2) taking the address of a global or stack-allocated variable using the `&` operator. At every `malloc()` call site, SoftBound inserts 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):

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;
```

For pointers to global or stack-allocated objects, the size of the object is known statically, so SoftBound inserts code to set the `base` to the pointer and `bound` to one byte past the end of the object:

```
int array[100];
ptr = &array;
ptr_base = &array[0];
ptr_bound = ptr_base + sizeof(array);
```

Pointer arithmetic and pointer assignment 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 `base` and `bound` of the original pointer:

```
newptr = ptr + index;    // or &ptr[index]
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

No checking is needed during pointer arithmetic because all pointers are bounds checked when dereferenced. As is required by C semantics, creating an out-of-bound pointer is allowed. SoftBound will detect the spatial violation whenever such a pointer is dereferenced. Array indexing in C is equivalent to pointer arithmetic, so SoftBound applies this same transformation to array indexing.

Structure field accesses Accesses to the fields of a structure are covered by the above transformations by conversion to separate pointer arithmetic and dereference operations.

Optional narrowing of pointer bounds The pointer-based approach adopted by SoftBound enables the ability to easily narrow the bounds of pointers, which in turn allows SoftBound to prevent internal object overflows. Shrinking of bounds can result in false violations for particularly pathological C idioms (discussed below), so SoftBound 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, SoftBound shrinks 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, SoftBound narrows the pointer's bounds to include only the individual field rather than the entire object.

```
struct { ... int num; ... } *n;
...
p = &(n->num);
p_base = max(&(n->num), n_base);
p_bound = min(p_base + sizeof(n->num), n_bound);
```

The above code 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:

```
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);
```

Although such narrowing of bounds may results in false positives, we have not encountered any false violations in any of our 23 benchmarks (approximately 272K lines of code). 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` [31]. Casts in SoftBound 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 SoftBound does 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 C++ `sort` function to sort a sub-array using `sort(&arr[4], &arr[10])`.

3.2 In-Memory Pointer Metadata Encoding

The above transformation only handled pointers as intermediate values (*i.e.*, values that can be mapped to registers). Pointers must also be stored to and retrieved from memory. SoftBound uses a table data structure to map an address of a pointer in memory to the metadata for that pointer. SoftBound inserts a table lookup to retrieve the base and bounds from the disjoint metadata space at every load of a pointer value:

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
newptr = *ptr;       // original load
newptr_base = table_lookup(ptr)->base;
newptr_bound = table_lookup(ptr)->bound;
```

Correspondingly, SoftBound inserts a table update for every store of a pointer value:

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
(*ptr) = new_ptr;     // original store
table_lookup(ptr)->base = newptr_base;
table_lookup(ptr)->bound = newptr_bound;
```

Only load and stores of pointers are instrumented; loads and stores of non-pointer value are unaffected. 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. The implementation section (Section 5) explores two implementations of the lookup table.

3.3 Metadata Propagation with Function Calls

When pointers are passed as arguments or returned from functions, their base and bound 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 above table-lookup approach for handling in-memory metadata would suffice. However, the function calling conventions of most ISAs specify that function arguments are generally passed in registers.

To address this issue, SoftBound uses procedure cloning [12] to transform every function declaration and function call site to include additional arguments for base and bound. For each pointer argument, base and bound arguments are added to the end of the list of the function's arguments. As part of this transformation, the function name is appended with a SoftBound-specific unique identifier, specifying this function has been transformed by SoftBound. For example, the code:

```
int func(char* s)
{ ... }
int value = func(ptr);
```

is transformed to:

```
int sb_func(char* s, void* s_base, void* s_bound)
{ ... }
int value = sb_func(ptr, ptr_base, ptr_bound);
```

Functions that return a pointer are changed to return a three-element structure by value that contains the pointer, its base, and its bound.

The transformation at the call site is performed entirely based on the arguments being passed to the function. Thus, this approach works when the definition and call site are in different files, which is necessary to support traditional separate compilation and external libraries. In fact, even if the function prototype is unspecified and incomplete (which is surprisingly common in actual C code), as long as the arguments passed in the C code are correct, the transformation will work as expected. This general approach has the additional benefit that the transformation is independent of the target ISA and the generated code obeys the system's standard calling conventions (albeit with additional parameters). Support of safe variable argument functions is discussed in Section 5.

3.4 Comparison with CCured's WILD Pointers

In many respects, SoftBound's pointer representation is merely a more compatible and more efficient implementation of CCured's WILD pointers. Like CCured's WILD pointers, SoftBound provides memory safety even in the context of arbitrary casts. CCured's WILD pointers accomplish this by (1) including a `base` field with each pointer, (2) including a size field at the beginning of each allocation, and (3) using tag bits at the end of each allocation to indicate which bytes in the allocation are pointers. These tag bits are written whenever storing to such an object (set to one when storing a valid pointer, set to zero otherwise) and read on every pointer load. As formally shown [35], this approach prevents corruption of the `base` pointer metadata stored inline within the objects, even if those objects are accessed via arbitrarily cast pointers. The key to this guarantee is that such stores will also set the tag to zero and that all pointer loads check this tag.

WILD pointers have three key disadvantages. First, WILD pointers introduce source code compatibility issues because they change memory layout in programmer-visible ways. Second, WILD pointer's `base` pointer must point to the start of an allocation, thus disallowing sub-object bounds information and failing to detect sub-object overflows. Third, all stores to a WILD object must update the metadata bits, adding runtime overhead. For these reasons (and the fact that WILD pointers disrupt CCured's whole-program type inference), all performance results for CCured are presented for benchmarks in which the need for WILD pointers was totally eliminated by program source modifications, program annotations, or insertion of unsafe trusted casts [35].

SoftBound's pointer representation improves upon WILD pointers while maintaining their spatial safety properties. First, SoftBound's metadata is recorded in a disjoint metadata space, avoiding the memory layout incompatibility of WILD pointers. Second, by using `base/bound` metadata that is totally decoupled from the pointer in memory, SoftBound avoids the object size header and tag bits, which in turn allows SoftBound to address the second deficiency of WILD pointers by allowing arbitrary sub-object bounding to detect sub-object overflows. Third, as SoftBound's metadata is disjoint, normal program memory operations cannot corrupt the metadata, eliminating both the tag bits and the need for every store operation to update these tag bits. With these improvements over WILD pointers, SoftBound pointer representation is highly compatible, provides reasonable performance overheads, and provides spatial safety even in the presence of arbitrary casts. The next section provides a formal proof that shows SoftBound's pointers provide the same well-formed memory guarantees (and thus spatial safety guarantees) as CCured's WILD pointers.

4. Formal Proof of Spatial Safety

This section sketches a safety proof for the key components of SoftBound's enforcement mechanisms, namely pointer metadata propagation and assertion checking. These claims have been mechanized using the Coq proof assistant [13]. A longer description of the proof and accompanying Coq source is publicly available [43].

Due to the size and complexity of the full SoftBound implementation, we concentrate our efforts on a fragment of C that covers covers almost all of SoftBound's features, including the address-of operator `&`, `malloc`, and named structure types, which permit recursive data structures. The formalism does not model SoftBound's checking of function pointers or capture its shrinking of bounds for detecting sub-object spatial violations.

At a high level, the proof has several steps. We first develop a non-standard operational semantics for (simplified, straight-line, and single-threaded) C programs that tracks information about which memory addresses have been allocated. To facilitate checking of spatial memory errors, our formalism axiomatizes properties of the C runtime with primitives for accessing memory: `read`, `write`, and `malloc`. As SoftBound targets spatial errors only, this formalism excludes `free` and other sources of temporal errors. Crucially, this semantics is *partial*—it is undefined whenever a bad C program would cause a spatial-safety violation; for programs without spatial memory errors, this semantics agrees with C.

Next, we augment the operational semantics so that it both propagates the bounds metadata and performs bounds-check assertions, aborting the program upon assertion failure. This step abstractly models the results of SoftBound instrumentation of the C program.

Finally, we define a well-formedness predicate on syntax that captures invariants ensured by a combination of the C compiler (*e.g.*, that local variables map to stack addresses) and SoftBound instrumentation. Standard preservation and progress results then establish that, starting from a well-formed initial program state, the SoftBound instrumented version will either terminate with a value, exhaust memory, or abort—it will never get stuck trying to access unallocated memory. This approach is similar to those used in both CCured's [35] and Cyclone's [24] formal developments. The rest of this section explains this proof strategy in more detail.

4.1 Syntax

The grammar in Figure 1 gives the fragment of C used in our proof. Commands consist of straight-line sequence of assignments, where the left-hand-side (*lhs*) is either a variable, a pointer dereference, or the field of a struct. The right-hand-side (*rhs*) of an assignment can be an integer constant, the result of an arithmetic operation, a *lhs*, the address of a *lhs*, the result of a cast, the size of a *pointer type*, or the result of `malloc`.

Atomic types are integers or pointers to *pointer types*, which include anonymous structure types, named structures and `void` in addition to atomic types. Here, *n* ranges over named structures, and *id* ranges over C identifiers. We assume that we have a partial map from names to anonymous structure types that represents `typedef`s in the source code.

4.2 Operational Semantics

The operational model is intended to represent programs *after* they have already been compiled to a fairly low level intermediate representation in which all code and data structures have been flattened and all operations are expressed in terms of atomic data types (ints and pointers). Our Coq proof defines a well-formedness predicate on syntax that picks out a subset of programs with these invariants, but we omit the details here and simply assume that all syntax mentioned in the rules below is well formed with respect to these invariants. Note that, because the compiler transformations that yield code in this intermediate form depend on source-program type information (to calculate struct field indices, for example) and SoftBound's instrumentation itself uses types, our non-standard semantics depends on source type information.

$$\begin{array}{rlcl}
\text{Atomic Types} & a & ::= & \texttt{int} \mid p\texttt{*} \\
\text{Pointer Types} & p & ::= & a \mid s \mid n \mid \texttt{void} \\
\text{Struct Types} & s & ::= & \texttt{struct\{}\cdots\texttt{;}id_i\texttt{:}a_i\texttt{;}\cdots\texttt{\}} \\
\text{LHS Expressions} & lhs & ::= & x \mid \texttt{*}lhs \mid lhs\texttt{.}id \mid lhs\texttt{->}id \\
\text{RHS Expressions} & rhs & ::= & i \mid rhs\texttt{+}rhs \mid lhs \mid \texttt{\&}lhs \\
 & & \mid & (a)\,rhs \mid \texttt{sizeof(}p\texttt{)} \\
 & & \mid & \texttt{malloc(}rhs\texttt{)} \\
\text{Commands} & c & ::= & c\texttt{ ; }c \mid lhs \texttt{ = } rhs
\end{array}$$

Figure 1. Syntax for the formal development.

Name	Specification
`read` *M l*	return `some` data at the address *l*, if accessible; `none` otherwise
`write` *M l* d	update data to *d* at the address *l*, if accessible; `none` otherwise
`malloc` *M i*	allocate a memory block with the size *i* if the free memory space is available; fail otherwise

Table 2. Memory operations.

The operational semantics for this C fragment also relies on an environment *E*, that has two components: A map, *S*, from variable names to their addresses and atomic types (modeling a stack frame), and a partial map, *M*, from addresses to values (modeling memory).

A memory *M* is defined only for addresses that have been allocated to the program by the C runtime. The C runtime provides three primitive operations for accessing memory: `read`, `write`, and `malloc`. Rather than committing to a particular implementation of these primitives, our formalism axiomatizes properties that any reasonable implementation should satisfy. Most of the axioms state simple properties like "reading a location after storing to it returns the value that was stored" and "storing to a location ℓ does not affect any other location." The most notable axioms involve `malloc`; they enforce properties like "malloc returns a pointer to a region of memory that was previously unallocated" and "malloc doesn't alter the contents of already allocated locations." Both `read` and `write` can fail if they try to access unallocated memory; `malloc` can fail if there is not enough space. Table 2 summarizes these operations.

Given these operations, we define a straightforward operational semantics for this fragment of C that is undefined for programs that access unallocated memory locations. The standard operational semantics of C evaluates a left-hand-side of an assignment to an address. The value at that address is overwritten by the value that the corresponding right-hand-side evaluates to. To model SoftBound's behavior, we augment this operational semantics to keep track of metadata and potential memory errors:

$$\text{Results} \quad r \quad ::= \quad v_{(b,e)} \mid l \mid \texttt{OK} \mid \texttt{Abort} \mid \texttt{OutOfMem}$$

A result *r* can be a value $v_{(b,e)}$ (the result of a right-hand-side) including the base (*b*) and bound (*e*) information along with the underlying data *v*. For example, if *v* is the value of a pointer to an array *array*, *b* and *e* are the start and end addresses of *array*; if *v* is the value of an integer, its base and bound information are zero. A result *r* can also be an address, *l* (the result of a left-hand-side), `OK` which is the the result of a successful command *c*, `Abort` when bounds check fails, or `OutOfMem` when memory allocation fails.

As mentioned above, to fully capture the instrumentation performed by SoftBound, it is also necessary for the operational semantics to propagate some type information. For example, before doing a dereference, SoftBound must do the bounds check according to pointers' base and bound information and the size of their types. Similarly, when casting integers to pointers SoftBound sets pointers' base and bound information to zero.

These considerations lead to three large-step evaluation rules. Left-hand-sides evaluate to a result r (which must be an address l if successful) and its atomic type a. Such an evaluation has no effect on the environment, so we write the rule as: $(E, lhs) \Rightarrow_l r : a$. Evaluating a right-hand-side expression also yields a typed result, but it may also modify the environment E via memory allocation or assignment, causing it to become E': $(E, rhs) \Rightarrow_r (r : a, E')$ (where r must be $v_{(b,e)}$ if successful). Commands simply evaluate to a result, which must be `OK` if successful, and final environment: $(E, c) \Rightarrow_c (r, E')$, where assignment statements can update the environment.

Space precludes showing the full set of operational rules (most of which are completely standard or obvious). Instead, we highlight a few cases that illustrate the salient features. For instance, the following examples show how to evaluate a pointer dereference when the bounds check succeeds (left) and fails (right):

$$\dfrac{\begin{array}{l}(E, lhs) \Rightarrow_l l : a* \\ \texttt{read}\,(E.M)\; l = \texttt{some}\; l'_{(b,e)} \\ l' \in [b, e - \mathrm{sizeof}(a)]\end{array}}{(E, *lhs) \Rightarrow_l l' : a} \qquad \dfrac{\begin{array}{l}(E, lhs) \Rightarrow_l l : a* \\ \texttt{read}\,(E.M)\; l = \texttt{some}\; l'_{(b,e)} \\ l' \notin [b, e - \mathrm{sizeof}(a)]\end{array}}{(E, *lhs) \Rightarrow_l \texttt{Abort} : a}$$

Here, lhs is first evaluated to l, which is the address of a pointer with type $a*$. If the address l is allocated, the function $\texttt{read}\,(E.M)\; l$ returns $l'_{(b,e)}$ the value of that pointer l' and its metadata b and e. Because pointers can be out-of-bounds due to pointer arithmetic, before doing the dereference, these rules check whether l' is within the bounds, and yield the error `Abort` when the bounds check fails. Note that it is a memory violation if $\texttt{read}\,(E.M)\; l = \texttt{none}$, in which case neither rule above applies. However, according to the type safety properties described below (Section 4.3), `read` will not fail at runtime.

As another example, the operational semantics performs bounds checking for access through a struct field using the following rule:

$$\dfrac{\begin{array}{ll}(E, lhs) \Rightarrow_l l : s* & \texttt{read}\,(E.M)\; l = \texttt{some}\; l'_{(b,e)} \\ l' \in [b, e - \mathrm{sizeof}(s)] & \texttt{getField}\; s\; id = \texttt{some}\,(\texttt{offset}, a)\end{array}}{(E, lhs\texttt{->}id) \Rightarrow_l l'\texttt{+offset} : a}$$

Here, `getField` returns a field offset, which is less than or equal to sizeof(s), along with the atomic type a of the field id in struct type s.

Other rules keep track of pointers' values and their metadata when evaluating pointer arithmetic (left) and type casts (right):

$$\dfrac{\begin{array}{l}(E, ptr) \Rightarrow_r (l_{(b,e)} : p*, E') \\ (E', i) \Rightarrow_r (n_{(b',e')} : \texttt{int}, E'') \\ l'\texttt{=}l\texttt{+}n\texttt{*}\mathrm{sizeof}(p)\end{array}}{(E, ptr\texttt{+}i) \Rightarrow_r (l'_{(b,e)} : p*, E')} \qquad \dfrac{\begin{array}{l}(E, rhs) \Rightarrow_r (v_{(b,e)} : a, E') \\ (b', e')\texttt{=}(a\texttt{==int})?(0,0) : (b, e)\end{array}}{(E, (a')rhs) \Rightarrow_r (v_{(b',e')} : a', E')}$$

As shown above, the metadata on the results of a pointer arithmetic operation is just the metadata associated with the original pointer. Casts also propagate the metadata unchanged, except in the case of casting an integer to a pointer, in which case the base and bound are both set to zero. Subsequent bounds checks on the resulting pointers will fail, ensuring that pointers created from integers cannot be dereferenced. This rule follows the SoftBound implementation, which associates metadata only with pointers stored in memory; as a consequence, when casting an integer to a pointer, the only sound choice is to zero-out the metadata. This approach is conservative, but in practice C programs rarely cast from pointers to integers and then back again, which is the case that might benefit from more accurate metadata propagation.

At runtime pointers can be temporarily out-of-bounds. The operational semantics will not yield an error in such cases until the program attempts a dereference through the illegal pointer. Once a bounds check fails or a memory allocation failure occurs, the rules propagate memory errors to the top level, analogously to raising an exception:

$$\dfrac{\begin{array}{l}(E, c_1) \Rightarrow_c (r, E') \\ r\ \textit{is}\ \texttt{Abort}\ \textit{or}\ \texttt{OutOfMem}\end{array}}{(E, c_1\ \texttt{;}\ c_2) \Rightarrow_c (r, E')} \qquad \dfrac{\begin{array}{l}(E, c_1) \Rightarrow_c (\texttt{OK}, E') \\ (E', c_2) \Rightarrow_c (r, E'') \\ r\ \textit{is}\ \texttt{Abort}\ \textit{or}\ \texttt{OutOfMem}\end{array}}{(E, c_1\ \texttt{;}\ c_2) \Rightarrow_c (r, E')}$$

4.3 Safety

The safety result relies on showing that certain well-formedness invariants are maintained by the SoftBound instrumented program. A well-formed environment $\vdash_E E$ consists of a well-formed stack frame S, which ensures that all variables are allocated in a valid memory block and have well-formed type information, and a well-formed memory M. A memory M is well formed when the metadata associated with each allocated location is well formed:

$$\dfrac{\forall l, d, b, e.(\texttt{read}\; M\; l = \texttt{some}\; d_{(b,e)}) \Rightarrow M \vdash_D\; d_{(b,e)})}{\vdash_M M}$$

$$\begin{array}{rl}M \vdash_D\; d_{(b,e)} \triangleq & (b = 0) \vee [(b \neq 0) \wedge\; (\forall l \in [b, e].\; \texttt{val}\; M\; l) \wedge \\ & (minAddr \leq b \leq e < maxAddr)]\end{array}$$

Here, $\texttt{val}\; M\; l$ is a predicate that holds when location l is allocated in memory M and $minAddr$ and $maxAddr$ bound the range of legal memory addresses where program data can reside. The judgment $\vdash_M M$ guarantees that if any address is accessible, its value is inbounds according to its metadata. A command is well-formed with respect to a stack frame S, written $S \vdash_c c$, when c typechecks according to standard C conventions assuming that each of the variables mentioned in c has the atomic type assigned by S.

With the above well-formedness conditions in place, Lemma 4.1 shows that if a left-hand-side evaluates to an address l without yielding any spatial memory violation, l must point to an allocated memory block that lies within the legal memory addresses range, and if a right-hand-side successfully evaluates to a value, its metadata represents a range of allocated memory.

LEMMA 4.1 (Successful Evaluation Ensures Safety).

1. *If* $\vdash_E E$, $(E, lhs) \Rightarrow_l l : a$, *then* $\texttt{val}\; E.M\; l \wedge minAddr \leq l \wedge l + sizeof(a) < maxAddr$.
2. *If* $\vdash_E E$, $(E, rhs) \Rightarrow_r (v_{b,e} : a, E')$, *then* $E'.M \vdash_D\; v_{(b,e)}$.

With Lemma 4.1, the type safety theorems show that SoftBound can detect memory violations at runtime.

THEOREM 4.1 (Preservation). *If* $\vdash_E E$, $E.S \vdash_c c$ *and* $(E, c) \Rightarrow_c (r, E')$, *then* $\vdash_E E'$.

THEOREM 4.2 (Progress). *If* $\vdash_E E$ *and* $E.S \vdash_c c$, *then* $\exists\; E'.\; (E, c) \Rightarrow_c (ok, E')$ *or* $\exists\; E'.\; (E, c) \Rightarrow_c (\mathrm{OutofMem}, E')$ *or* $\exists\; E'.\; (E, c) \Rightarrow_c (\mathrm{Abort}, E')$.

The proofs of these theorems are straightforward inductions on the structure of the typing derivations, and the type safety properties of lhs expressions and rhs expressions which also follow by inductions on the structure of the typing derivations. These theorems also imply the following corollary:

COROLLARY 4.1 *If* $\vdash_E E$, $E.S \vdash_c c$ *and* $\exists\; E'.\; (E, c) \Rightarrow_c (ok, E')$, *then the original C program will not cause any spatial memory violation.*

5. Implementation

The previous sections have described SoftBound's basic approach and formal justification. This section describes the specific implementation of SoftBound's metadata facility and specifically how SoftBound handles various aspects of C (global variables, separate compilation, `memcpy()`, function pointers, arbitrary casts, and variable argument functions).

5.1 Metadata Facility Implementation

SoftBound's metadata facility completely decouples the metadata from the pointers in memory. At its simplest, SoftBound must map an address to the base and bound metadata for the pointer at that address (*i.e.*, the lookup is based on the location being loaded or stored, not the value of the pointer that is loaded or stored). This mapping can be implemented in several ways, including lookup trees or tables; SoftBound uses table lookup.

As the metadata accesses can be a significant source of runtime and memory overhead, we explore two implementations of the metadata facility: a hash table and a tag-less shadowspace. Each organization comes with its own set of trade-offs with respect to memory and performance overheads.

Hash table The conceptually most straightforward implementation of the metadata facility is a simple hash table. Each entry in the table has three entries: tag, base, and bound. Assuming 64-bit pointers, each entry is 24 bytes. To reduce runtime overhead, this implementation uses a simple hash function: the double-word address modulo the number of entries in the table. By keeping the number of entries in the table a power of two, this calculation is a simple shift-and-mask operation. Collisions are handled using open hashing. Collisions are minimized by sizing the table large enough to keep average utilization low. In the common case of no collisions, the lookup is approximately nine x86 instructions: shift, mask, multiply, add, three loads, tag compare, and branch.

Shadow space The shadow space implementation reduces the overhead of the hash table by allocating a large enough table in the virtual address space such that collisions are guaranteed not to occur. With this guarantee, the tag field and checking is eliminated, reducing both worst-case memory overhead and instruction count. A shadow space lookup is approximately five x86 instructions: shift, mask, add, and two loads.

To ensure no collisions occur, the stack and heap are each limited to the top and bottom eighth of the virtual address space, respectively. The system reserves a large region of memory in the middle of the virtual address space for the shadow space. In essence, this approach reduces the size of the virtual address space by two bits. The SoftBound prototype uses `mmap()` to create a zero-initialized region in virtual memory, and the operating system then allocates physical pages for this region on demand (*i.e.*, when each page is first accessed).

5.2 Implementation Considerations

Handling arbitrary C programs requires addressing several issues.

Global variables For global arrays, the base and bound are compile-time constants. Thus, SoftBound 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, SoftBound adds code to explicitly initialize the in-memory base and bounds for these variables. This can be implemented using the same hooks C++ uses to run code for constructing global objects.

Separate compilation and library code Unlike proposals that exploit whole-program analysis [17, 35], SoftBound easily supports separate compilation. As described earlier, SoftBound uses procedure cloning [12] to transform functions to take additional parameters and changes the function name to signify that it has been transformed. Separate compilation works naturally—the static or dynamic linker matches up caller and callee as usual. This support for separate compilation has two important ramifications. First, SoftBound supports build environments in which a large program is built by compiling many distinct modules via separate compilation. Second, SoftBound can be applied directly to library code, allowing a library writer to create and distribute a single library archive with both transformed (spatially safe) and untransformed (unsafe) versions of each function. For libraries that have not (yet) been transformed, SoftBound employs library function wrappers similar to those used in MSCC [47] or CCured [35] (but without the marshaling issues caused by incompatible memory layout).

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 bounds safety once at the start of the copy. Second, `memcpy` must also copy the metadata corresponding to any pointer in the region being copied. SoftBound could take the safe (but slow) approach of always inserting code to copy the metadata, yet most calls to `memcpy` involve buffers of non-pointer values. To address this inefficiency, SoftBound infers whether the source of the `memcpy` contains pointers by looking at the type of the argument at the call site. Although not foolproof, we have found this heuristic sufficient to identify the few uses of `memcpy` involving pointers in our benchmarks.

Function pointers To protect function pointers, the SoftBound prototype sets the base and bound for function pointers to be equal to the pointer. Such an encoding is not used by data objects (it would correspond to a zero-sized object), so SoftBound can check for this metadata when the program calls through a function pointer. Although this encoding prevents data pointers or non-pointer data from being interpreted as a function pointer, casts between function pointers of incompatible types presents a challenge because calling a function with arbitrary values may allow the manufacture of improper base and bounds. Though not yet implemented in our prototype, to ensure complete protection, a full implementation would encode the pointer/non-pointer signature of the function's arguments, allowing a dynamic check to properly handle such cases.

Creating pointers from integers By default SoftBound sets the base and bound of a pointer created from a non-pointer value to NULL. This is a safe default (any dereference of such a pointer will trigger a bounds violation), but may cause false violations in particularly perverse C programs. Although we have not encountered such code in the applications we have examined, SoftBound supports such casts by providing a `setbound()` function that allows the programmer to bypass safety guarantees by explicitly setting the bound for a pointer (including completely "unbounding" a pointer to provide unchecked access to arbitrary memory locations if the programmer so desires). The programmer may also use the `setbound()` function to explicitly shrink bounds, for example, when employing a custom memory allocator.

Arbitrary casts and unions C supports arbitrary type conversion by explicit casts and implicit conversions via unions. SoftBound allows all such casts, because separating the metadata and program data ensures that pointer bounds are not unsafely manipulated by casts. In contrast, inline fat pointer schemes [5, 35, 47] have difficulty supporting arbitrary casts. In SoftBound, casts among pointer types simply inherit the same bounds. Like CCured's WILD pointers, SoftBound enforces spatial memory safety as defined by well-formed memory invariants (Section 4.3). That is, SoftBound ensures that a pointer can only dereference memory locations within its bounds, and that all those memory locations are valid; it does not provide an assurance about the types of those memory locations.

Variable argument functions Variable argument functions are another source of unsafe behavior of C programs. An approach to handling variable arguments functions is to introduce a safer calling convention for variable argument functions by including two extra parameters: the number of parameters passed (in bytes) and the number of pointers passed (and thus the number of extra base/bound arguments passed to the function). The `va_start` and `va_arg` macros could then check that neither too many arguments nor too many pointer arguments are decoded. Our SoftBound prototype does not yet implement this safer calling convention for variable argument functions.

6. Experiments

In this section, we describe and experimentally evaluate our prototype implementation of SoftBound. The goal of this evaluation is to (1) show SoftBound is effective in preventing spatial violations, (2) measure its runtime and memory overheads, and (3) to show SoftBound is compatible with existing C code.

6.1 The SoftBound Prototype

The SoftBound prototype uses LLVM [33] version 2.4 as its foundation. SoftBound operates on LLVM's fully typed single static assignment (SSA) intermediate form. LLVM invokes the SoftBound pass after it has performed its full set of optimizations. Applying SoftBound post-optimization ensures that SoftBound's instrumentation does not prevent code optimization. Furthermore, as register promotion and other optimizations have already reduced the number of memory operations, this strategy reduces the amount of additional instrumentation introduced by SoftBound.

The SoftBound pass inserts code to (1) create a base and bound value for each pointer non-memory value in the program, (2) perform base/bound metadata manipulation prior to every memory operation that reads or writes a pointer, (3) perform a bounds check before memory operations and (4) rewrite all function calls to pass the base and bounds as was described in Section 3. To eliminate some obviously redundant checks of the same pointer, our prototype performs a simple intra-procedural dominator-based redundant check elimination. These transformations are all strictly local (intra-procedural) transformations, without using any whole-program type inference or alias analysis. Calls to external functions (*i.e.*, any library function that has not been SoftBound transformed) are mapped to wrapper functions.

SoftBound uses standard C functions to implement the code to access the base/bound metadata and to perform the bounds checks. The SoftBound pass invokes these routines by inserting appropriate function calls that are later forcibly inlined by subsequent LLVM passes.

After SoftBound has transformed the intermediate code, it reruns the full suite of LLVM optimizations on the instrumented code. This simplifies the SoftBound pass, because subsequent optimization passes will remove dead code and factor out common sub-expressions. To reduce compilation time in production environments, SoftBound would likely become an internal pass performed after early optimizations such as register promotion and function inlining, but before the most time-consuming optimization passes.

The SoftBound pass operates on LLVM's ISA-independent intermediate form, so the SoftBound pass is independent of any specific ISA. We selected 64-bit x86 as the ISA for evaluation due to its ubiquity. The SoftBound pass is approximately 5000 lines of C++ code, and we plan to make the SoftBound source code publicly available [43].

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. Various synthetic attacks proposed by Wilander *et al.* [45] and SoftBound's ability to detect them with full checking and store-only checking.

6.2 Effectiveness in Preventing Vulnerabilities and Bugs

To evaluate the effectiveness of SoftBound in detecting violations of spatial safety, we applied SoftBound to a suite of security violations [45] and to versions of programs with well-documented security violations [34]. SoftBound detects all the spatial violations and prevents all the security vulnerabilities in these tests without any false positives.

We use a testbed of buffer overflow attacks [45] that includes overflows on the stack, heap, and global segments to overwrite various return addresses, data pointers, function pointers, and longjmp buffers. Table 3 lists the attacks based on the technique adopted, location of the overflow, and the attack target that is used to change the control flow. SoftBound detects and prevents all these attacks in both full and store-only checking mode. Publicly available tools such as StackGuard, ProPolice, Libsafe and Libverify miss more than 50% of these test cases [45].

We also evaluated SoftBound's ability to detect spatial bugs using actual spatial errors from real programs obtained from the BugBench suite [34]: go, compress, gzip, and polymorph. These bugs are a mixture of one or more read or write overflows on the heap, stack, and globals. Table 4 lists the benchmarks and the detection ability of SoftBound and various memory checking tools. SoftBound with full checking was able to detect and prevent all of the errors. SoftBound with checking only for stores was able to detect all of the store overflows, but not the load overflows.

As a point of comparison, Table 4 also reports the efficacy of memcheck [42] and ptrcheck [36] from version 3.4.1 of Valgrind, Mudflap [21] from GCC 4.2.0, and the Jones and Kelly [29] modification to version 4.0.0 of GCC. Like SoftBound, 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

	Violation Detected?					
	SoftBound		Valgrind		GCC's	J&K
Benchmark	Store	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 4. Programs with overflows and the detection efficacy of SoftBound (store-only and full checking), Valgrind's memcheck [42], Valgrind's ptrcheck [36], GCC's Mudflap [21], and Jones and Kelly [29].

to detect violations that SoftBound detects. For example, Valgrind's memcheck tool does not detect overflows on the stack, leading to its failure to detect some of the bugs.

6.3 Runtime Overhead Evaluation

Benchmarks We used 23 benchmarks selected from the SPECint, SPECfp, and Olden benchmark suites to evaluate SoftBound's performance. The Olden benchmarks [40] were used because they have been used in the most significant prior work in this area [17, 35, 47]. Our SoftBound prototype is not yet robust enough to compile all of the C programs in the SPEC benchmark suites, but we present data for the benchmarks we examined for which SoftBound works correctly. All runs are performed on a 2.66 Ghz Intel Core 2 processor. Hardware performance counters were used to measure dynamic instruction count and cache effects.

Benchmark characterization One of the main sources of overhead in SoftBound is the runtime overhead of metadata accesses, which is highly dependent on the frequency of such accesses. Our experiments show that the frequency of metadata accesses varies significantly from benchmark to benchmark. Figure 2 shows the benchmarks sorted by the percentage of memory operations that load or store a pointer value. The SPEC benchmarks are dark bars; Olden benchmarks are white bars. Several of the benchmarks have metadata access ratios of less than 10%, including eleven of the sixteen selected SPEC benchmarks. In the other extreme, over half of the memory operations in several of the Olden benchmarks are loads and stores of pointers. To more easily show the correlation of metadata accesses and runtime performance, the remaining graphs will present the benchmarks in this sorted order.

Runtime overheads of full checking Figure 3 presents the percentage of runtime overhead of SoftBound over an uninstrumented baseline (smaller bars are better as they represent lower runtime overheads). This graph contains a pair of bars for each benchmark. The total height of each bar corresponds to the overhead of full checking using the hash table (left bar of each pair) and shadow space implementations (right bar of each pair) of the metadata facility. The average runtime overhead is 93% (for the hash table implementation) and 67% (for the shadow space implementation). For all the benchmarks these runtimes overheads are likely more than acceptable for debugging, internal and external testing, and for mission-critical applications.

The benchmarks on the left of the graph (those with a lower frequency of metadata accesses) generally have lower runtime overheads. On those benchmarks the overhead is largely due to the actual checking of the bounds, so the specific metadata encoding scheme has little impact on the overhead. The SoftBound prototype's simple dominator-based redundant check elimination does reduce the overheads of bounds checking—it reduces the overall average runtime overhead from 80% to 67%—but more advanced bounds check elimination techniques (*e.g.*, [8, 46]) would further reduce the overheads.

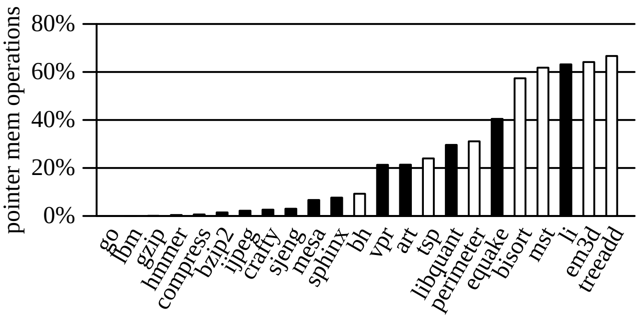

Figure 2. The percentage of memory operations that load or store a pointer from/to memory, thus requiring a metadata access. Benchmarks from SPEC are represented by shaded bars; Olden benchmarks are represented by white bars.

In contrast to the benchmarks on the left whose overhead was primarily dominated by bounds checks, the benchmarks on the right of the graph perform frequent metadata accesses. For these benchmarks, metadata accesses are a large source of overhead, and they are significantly impacted by the metadata encoding. The shadow space encoding outperforms the hash table encoding, sometimes substantially. For a few benchmarks (`equake`, `li`, and `em3d`), the shadow space reduces the runtime overhead by more than half.

Runtime overhead of store-only checking Our experiments with the security vulnerabilities reinforce the intuition that checking only stores can prevent security vulnerabilities. Moreover, in our experience, store overflow bugs are more 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. The height of just the bottom segment (stripe pattern) of each bar in Figure 3 represents the overheads for checking the bounds only for store operations. Checking only stores reduces the number of check operations. However, the number of metadata writes is unchanged, because all the same pointer metadata must be propagated through memory to check subsequent store dereferences. When using the shadow space implementation, LLVM's dead code elimination also removes many of the metadata reads (those that feed only load dereference checks). However, as the hash table implementation uses a loop, LLVM is unable to remove the metadata reads. As a result, in the current SoftBound prototype, store-only checking benefits the hash table implementation less than the shadow space implementation, for average runtime overheads of 54% and 22%, respectively. Furthermore, the runtime overhead of store-only checking using the shadow space is less than 15% for more than half of the benchmarks, which is likely low enough for production code.

Dynamic instruction count overhead Figure 4 is similar in form to the previous figure, except it shows the dynamic instruction count overheads (rather than execution time overhead) for the various configurations. This graph confirms that accessing the hash table shadow space executes more instructions than the shadow space implementation. In most cases, this larger number of instructions is the primary source of additional overhead of the hash table implementation.

The dynamic instruction count overheads are highly correlated with the runtime overheads, but the dynamic instruction count is generally much larger than the corresponding runtime overhead. On average SoftBound's full checking increases the instruction count by 173% (hash table) and 133% (shadow space), which is almost

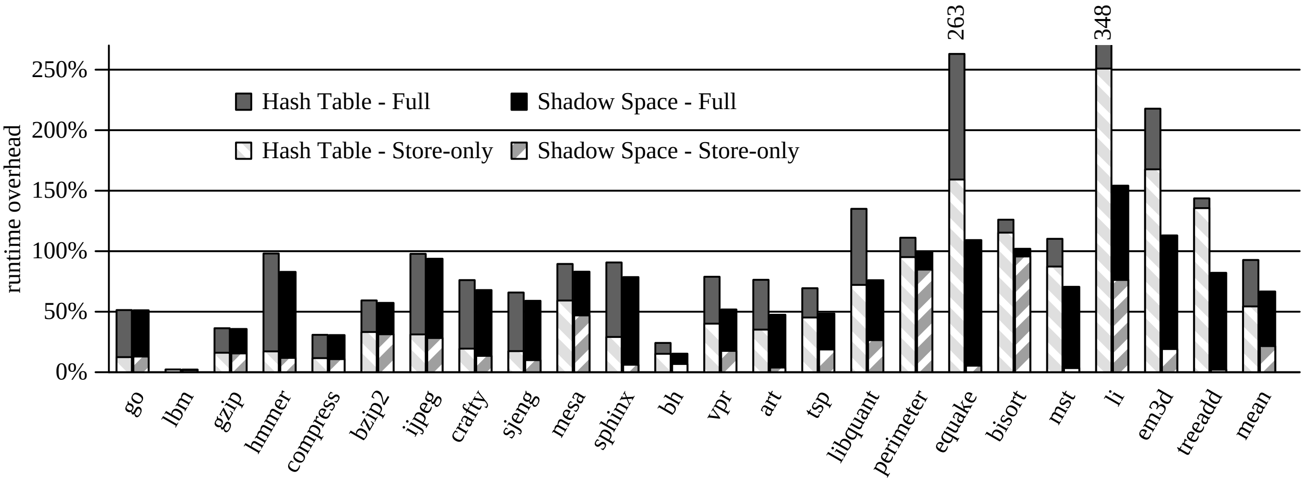

Figure 3. Normalized execution time overhead of SoftBound with full checking and store-only checking with two metadata organizations.

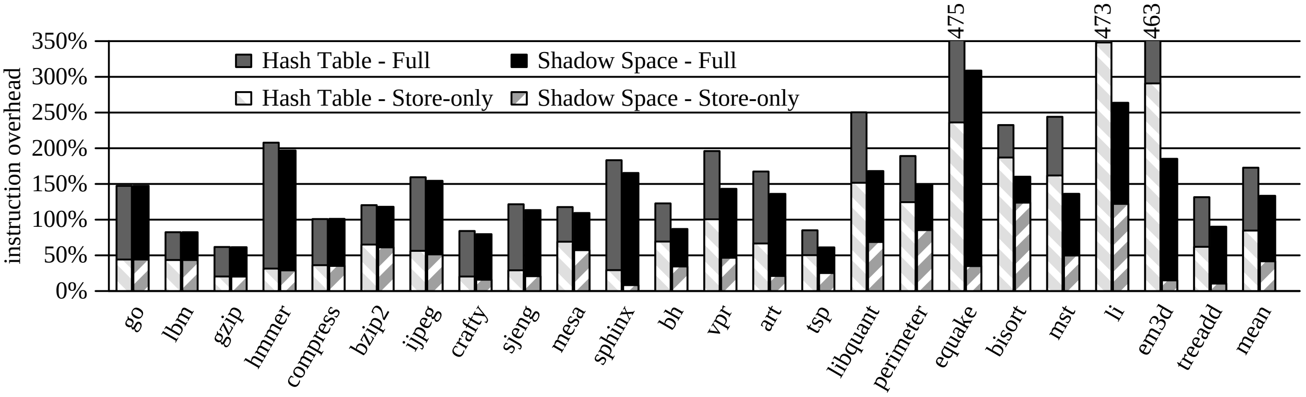

Figure 4. Dynamic instruction overhead of SoftBound.

double the runtime overheads. This data implies that the *instruction per cycle* (IPC) is higher when executing code instrumented by SoftBound.

In essence, SoftBound's metadata manipulation and bounds checking add to the instruction count, but they also increase the available *instruction-level parallelism* (ILP). The dataflow subgraph of base and bound metadata manipulations is disjoint from the dataflow graph of the original program computation. These manipulations are also not on the control-flow critical path of the program's execution, assuming the processor correctly predicts both (1) the never-taken branch that is part of the bounds check and (2) the rarely-taken branch as part of the hash table tag check. The Intel Core 2 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 four micro-operations per cycle. As few programs have enough ILP to sustain four-wide execution, some of the instructions added by SoftBound are executed "for free" by the unused execution capacity. For example, hardware performance counters report that the benchmark `lbm` has a high data cache miss rate of one miss every 20 instructions. The resulting low IPC (just 0.22) provides plenty of spare execution capacity to hide SoftBound overheads, which explains `lbm`'s runtime overhead of only 3% while its instruction execution overhead is 82%.

This finding has several ramifications. First, today's sophisticated processors are partly responsible for the low overheads exhibited by SoftBound. Second, the runtime overhead of spatial safety enforcement will be higher when running on low-cost power-efficient processors commonly used in mobile devices and embedded systems. Third, if SoftBound or any of the other proposals for enforcing spatial safety become widely adopted, it could significantly impact the tradeoffs in the design of microprocessors. Conversely, future microprocessor designs could significantly positively or negatively impact the runtime overheads of enforcing spatial safety for C programs.

Memory overheads One cost of SoftBound is that its disjoint metadata increase the program's memory footprint. Figure 5 shows the normalized memory overheads based on the total number of 4KB pages touched during the entire execution for SoftBound with both the hash table and the shadow space encoding. For programs with many linked data structures (and thus many in-memory pointers), the worst-case memory footprint overhead can be as high as 300% for the hash table (one 24-byte entry for each 64 bits of memory) or 200% for the shadow space (one 16-byte entry for each 64 bits of memory). However, the memory overhead for many benchmarks is much lower. The average memory overheads are 87% and 64%, respectively. The hash table in these experiments is sized to be large enough maintain metadata for every allocated memory location with few collisions in the table, effectively trading space to reduce instruction count overheads. A smaller hash table would have the reverse effect.

Cache effects Another impact of SoftBound's metadata is additional cache misses and cache capacity pressure. Figure 5 plots the

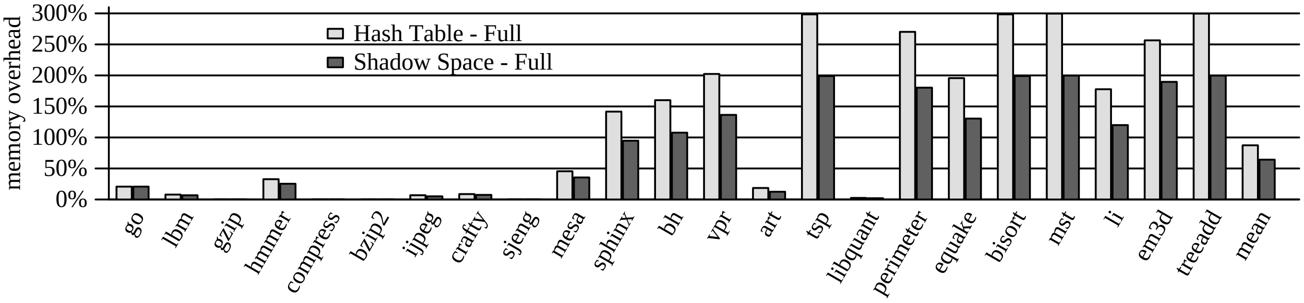

Figure 5. Memory footprint overhead for various benchmarks

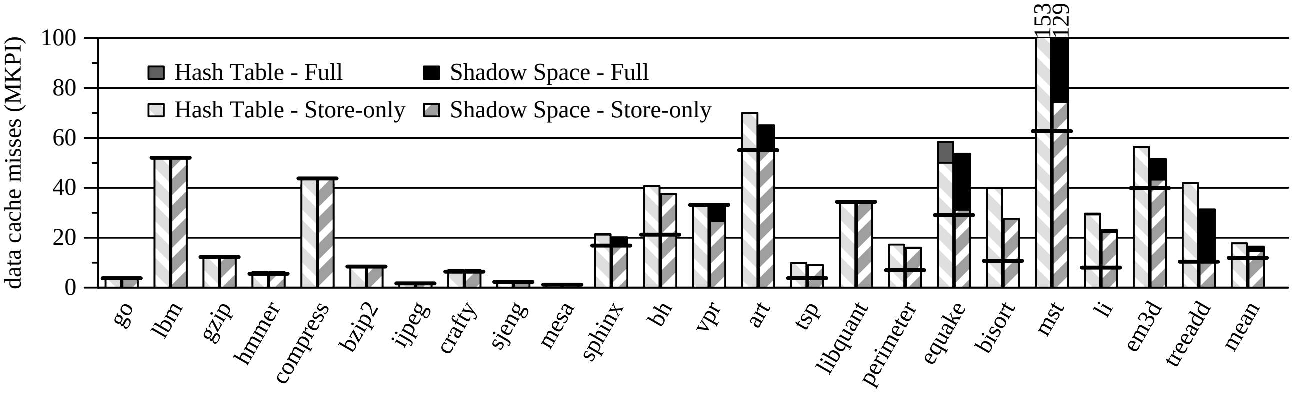

Figure 6. Data cache misses per 1000 base instructions for various benchmarks.

miss rate of the processor's 32KB first-level data cache. The miss rates are given as misses for every thousand instructions executed *by the baseline non-SoftBound execution*, which prevents differences in the number of dynamic instructions executed by the four SoftBound configurations from obfuscating the data. The horizontal line across each pair of bars represents the miss rate of the baseline execution. The benchmarks with few metadata accesses (those on the left of the graph) show no difference in the baseline misses and the misses for the SoftBound configurations (*i.e.*, the horizontal line is at the top of the stack of bars). For the benchmarks with more frequent metadata accesses (those on the right side of the graph) show an increase in the number of cache misses. In some cases, the miss rate more than doubles (*e.g.*, `bh`, `bisort`, `mst`, `li`). Unsurprisingly, the benchmarks with high baseline miss rates and with significantly more misses also have high runtime overheads. The hash table implementation shows little difference in miss rates between full and store-only checking, because few metadata loads are eliminated (as was discussed above).

6.4 Source Code Compatibility Case Studies

To evaluate our claim that SoftBound is highly compatible with existing source code and interfaces well with existing libraries, we applied SoftBound to two network server applications: a fully-functional FTP server (tinyftp-0.2) and high-performance web server with CGI support from NullLogic (nhttpd-0.5.1). The NullLogic HTTP server is multithreaded and capable of handling thousands of simultaneous connections. SoftBound successfully transformed these network applications without requiring any source code modifications and no false positives during program execution. Apart from these network applications, SoftBound also successfully transformed the 23 benchmarks used in the performance evaluation. In total, these benchmarks and network servers are approximately 272K total lines of code, all of which were successful transformed, further supporting SoftBound's source code compatibility claim.

6.5 Performance Comparison to Related Approaches

Table 5 reports the overhead of two publicly available compiler-based object-based implementations. The overheads are generally much higher than SoftBound's overhead: 12.1x, 24.1x and 7.7x for Jones and Kelly, Mudflap with full checking and MudFlap with store only checking respectively. Table 5 also includes the overheads of two memory checking tools Valgrind, which are based on dynamic binary instrumentation.

CCured [35] and MSSC [47] are two pointer-based schemes closely related to SoftBound. CCured has low runtime overheads, ranging from 3% to 87% [35]. CCured's whole-program type inference statically removes many metadata manipulations, resulting in overheads that are lower on average than SoftBound. However, on benchmarks whose overhead is dominated by dereference bounds check overhead (*e.g.*, the SPEC benchmark `compress`), SoftBound and CCured have similar overheads. Furthermore, applying CCured to a program requires non-trivial changes to the source code. Although some of the changes are simple, restructuring a program to avoid all casts that cause WILD pointers may require extensive code changes such as runtime type information annotations and tagged unions —or ultimately giving up on complete safety by marking casts as trusted [35]. Lu *et al.* used CCured to investigate its bug detection ability and have described these code modifications as "moderate" to "hard" and ultimately failed to apply CCured to one benchmark [34].

MSCC [47] has higher overheads than CCured, partly because it eschews whole-program analysis (as does SoftBound). When configured to perform only spatial safety checking, MSCC's overheads

Benchmark	J&K	Mudflap		Valgrind	
		Store	Full	memcheck	ptrcheck
go	28.3x	300.4x	316.0x	19.8x	98.5x
lbm	4.8x	1.4x	1.9x	4.6x	12.0x
gzip	9.9x	2.1x	2.8x	19.1x	86.1x
hmmer	—	2.5x	4.8x	13.8x	86.4x
compress	14.1x	1.5x	3.5x	12.8x	54.1x
bzip2	17.3x	8.9x	10.3x	13.8x	75.0x
ijpeg	40.1x	69.4x	71.1x	16.2x	95.3x
crafty	13.9x	166.9x	170.2x	41.3x	209.2x
sjeng	25.7x	19.8x	19.8x	27.3x	27.3x
mesa	31.8x	193.9x	197.0x	35.5x	154.0x
sphinx	24.8x	4.7x	43.8x	24.4x	61.5x
bh	32.4x	—	—	14.4x	62.7x
vpr	25.5x	2.5x	9.6x	21.9x	130.1x
art	242.4x	2.9x	91.1x	16.9x	49.2x
tsp	9.0x	2.7x	15.9x	24.2x	93.7x
libquant	35.5x	3.0x	118.5x	12.7x	66.7x
perimeter	10.8x	6.1x	13.1x	14.9x	73.6x
equake	4.5x	28.4x	208.1x	13.3x	70.7x
bisort	18.3x	23.3x	32.5x	9.4x	37.0x
mst	22.4x	1.9x	11.2x	3.8x	11.1x
li	22.7x	20.0x	20.0x	27.8x	176.1x
em3d	—	6.7x	40.1x	13.1x	58.6x
treeadd	7.5x	7.1x	756.4x	44.2x	—
Average	12.1x	7.7x	24.1x	16.7x	65.8x

Table 5. Runtime overhead of Jones & Kelly [29], Mudflap [21] with store only and complete checking, Valgrind's memcheck [42] and, Valgrind's ptrcheck [36]. Entries with "—" indicate an internal compiler error, spurious runtime exceptions, or trials discarded because of extremely long runtimes.

range from 15% to 185% with an average overhead of 68% [47]. Our own experimentation with MSCC and the published results indicate that SoftBound's overhead is generally similar to or somewhat lower than MSCC's overhead on common benchmarks.

7. Additional Related Work

Many other approaches other than enforcing full spatial safety have been explored for detecting and diagnosing bounds violations or preventing bounds-related security vulnerabilities. Many static analyses that detect buffer overflows have been proposed, including using abstract interpretation [7, 20] and integer programming [22]. Static analysis has also been coupled with lightweight programmer or inferred annotations (*e.g.*, [11, 26]). Static checking tools generally either have false positives or false negatives (they are incomplete), but are certainly useful complementary techniques to dynamically enforced spatial memory safety.

Other approaches monitor control flow transfers [30], ensure control flow integrity [3], or enforce dataflow integrity based on reaching definition analysis calculated statically [10]. Pointer analysis can also be used to compute the approximate set of objects written by each instruction [4]. In all four cases, these properties are checked dynamically, but neither strategy directly enforces memory safety. Probabilistic memory safety approaches, such as DieHard [6], prevent many security vulnerabilities in the heap by using a randomized runtime system and achieving probabilistic memory safety by approximating to an infinite size heap.

8. Conclusion

SoftBound is a compile time transformation system to provide spatial safety for the C programming language without changes to the source code. SoftBound accomplishes this using a pointer-based approach with a disjoint metadata space. Further, the mechanized formal proof shows SoftBound's metadata propagation is sufficient to provide spatial safety even for programs with arbitrary casts.

We experimentally verified SoftBound's ability to catch spatial violations using real benchmarks with overflows and a suite of security vulnerabilities. We found that SoftBound successfully transformed several benchmarks and two network daemons (around 272K lines of code total) with no source code modifications. SoftBound's performance overhead is 67% and 22% on an average in its full and store-only checking modes, respectively. SoftBound's store-only checking mode has less than 15% overhead for more than half of the benchmarks. This runtime overhead is likely low enough to be employed in production code, giving SoftBound the potential to substantially improve the security and robustness of real-world software systems.

Acknowledgments

The authors thank Vikram Adve, Joe Devietti, Andy Glew, Dan Grossman, and Jeff Vaughan for comments on this work. We thank the LLVM developers for answering our LLVM questions, Shan Lu for the BugBench code, John Wilander for the dynamic security testbench, Nachiket Kapre for help with hardware performance counters, and Wei Xu for help with MSCC. This paper is based upon work supported in part by NSF (National Science Foundation) awards CNS-0524059, CCF-0644197, and CCF-0810947, Defense Advanced Research Projects Agency (DARPA) contracts RA06-46 and FA8650-07-C-7743, and donations from Intel Corporation. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, DARPA, or Intel.

References

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

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

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Nov. 2005.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.2beta4)*, 2008.

[14] 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.

[15] 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.

[16] 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 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] D. Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Department of Computer Science, Cornell University, Aug. 2003.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

[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] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

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

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.