Department of Computer & Information Science

Technical Reports (CIS)

University of Pennsylvania *Year* 2009

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte* Jianzhou Zhao†
Milo Martin‡ Stephan A. Zdancewic**

*University of Pennsylvania
†University of Pennsylvania
‡University of Pennsylvania, milom@cis.upenn.edu
**University of Pennsylvania, stevez@cis.upenn.edu

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

Technical Report MS-CIS-09-01 — January 2009

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 complete spatial safety of C. SoftBound records base and bound information for every pointer as disjoint metadata. This decoupling enables SoftBound to provide complete spatial safety while requiring no changes to C source code. Moreover, SoftBound performs metadata manipulation only when loading or storing pointer values. A formal proof shows this is sufficient to provide complete spatial safety even in the presence of wild casts. SoftBound's full checking mode provides complete spatial violation detection. To further reduce overheads, SoftBound has a store-only checking mode that successfully detects all the security vulnerabilities in a test suite while adding 15% or less overhead to half of the benchmarks.

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. [1] Worse yet, it is the underlying root cause of a multitude of security vulnerabilities [28, 31]. 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 example, in November, 2008 Adobe released a security update that fixed several serious buffer overflows [2]. Attackers have reportedly exploited such buffer overflow vulnerabilities by using banner ads to redirect users to a malicious PDF document crafted to take complete control of the victim's machine [1].

Safe languages such as Java and C# completely prevent this entire class of bugs and security vulnerabilities by enforcing memory safety. 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, thus preventing the 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, 9, 13, 18, 19, 23, 26, 30, 34, 35]. Unfortunately, several aspects of C, such as its conflation of arrays and singleton pointers, unchecked array indexing and pointer arithmetic, pointers to the middle of objects, arbitrary casts, user-visible memory layout, and structs with internal arrays all interact to greatly increase the difficulty of retrofitting C with spatial memory safety. As a result, these proposals all suffer from one or more practical difficulties that may prevent wide adoption, such as: unacceptably high runtime overheads, incomplete detection of spatial violations, incompatible pointer representations (by changing memory layout), or requiring non-trivial changes to existing C source code. Moreover, the proposals with the lowest performance overheads generally employ whole-program compiler analyses (*e.g.*, [13, 23]) which complicates separate compilation and use of dynamically linked libraries. Section 2 provides additional background on these proposals.

This paper describes SoftBound, a compile-time transformation for inserting runtime bounds checks to enforce complete spatial safety of C programs. SoftBound uses a pointer-based approach to enforce spatial safety, by associating base and bound metadata with every pointer. Unlike prior pointer-based approaches that change pointer representations and thus object layouts [4, 23, 34], 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. SoftBound performs a simple intra-procedural transformation that instruments and renames each function to propagate and check pointer metadata. Functions with pointer arguments or return values are extended with additional parameters for base and bound metadata. This 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 several benchmarks and two network daemons show that SoftBound can be successfully applied to unmodified C programs.
- **Completeness** By default the SoftBound tranformation guarantees complete 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 [23]. Our experiments show SoftBound catches errors not caught by Valgrind [24] or GCC's Mudflap [15].
- **Separate Compilation** SoftBound's simple intraprocedural analysis, disjoint metadata, and function renaming provides

[1] Temporal safety violations (*i.e.*, dangling pointers) are also a significant source of bugs in C programs. As spatial safety is the primary concern for security vulnerabilities, this works focuses exclusively on the spatial safety issues of C. Other previously proposed complementary techniques such as conservative garbage collection, reference counted safe pointers, or probabilistic approaches may be employed to detect temporal violations.

transparent support for separate compilation, allowing library code to be recompiled with SoftBound and dynamically linked. This extends checking into library code and can avoids 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 complete safety at the cost of a 79% runtime overhead on average over a range of 15 benchmarks. In contrast to heavyweight instrumentation [24] used selectively during the development process, 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, SoftBound provides a store-only checking mode. In this mode, SoftBound fully propagates all metadata, but inserts bounds checks only for memory writes. As observed previously [3], 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 detects all the vulnerabilities in a security vulnerability suite at that cost of 32% average overhead. The 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 as debugging aids [4, 19, 27], but have now been improved immensely [11, 13, 18, 23, 30, 34]—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 handling spatial violations, this section does not discuss approaches for preventing temporal safety violations (*i.e.*, dangling pointers). Other less directly related work is discussed in Section 7.

2.1 Object-based Approaches

Object-based approaches [11, 13, 15, 19, 30] track all object allocations in a separate data structure to allow any addresses to be mapped to a specific object. All pointer operations, including arithmetic and dereference, are checked to ensure that they remain within the bounds of the same object. When out-of-bound pointers do occur, a special out-of-bounds object is used [13, 19]. When such an out-of-bound pointer is modified so that it is back in bounds, the enforcement mechanism must ensure that it points back to the original object. The important advantage of the object-pointer approach is that memory layout of objects is unchanged, providing source and binary compatibility.

However, object-based schemes have two disadvantages. First, the object-lookup table is often implemented as a splay tree, which can be a performance bottleneck, yielding runtime overheads of 5x or more [15, 19, 30]. Subsequent proposals have considerably mitigated this issue by reducing the overhead of object-table-based approaches by checking only strings [30] or using whole-program analysis to perform automatic pool allocation to partition the splay trees and eliminate lookup and checking of many scalar objects [11, 13].

The primary drawback of the object-based approaches is that are incomplete. That is, they do not detect all spatial violations, for example, arrays inside structures are not checked [13, 19]. Consider the example below

```
1. struct { char str[8]; void (*func)();} node;
2. char* ptr = node.str;
3. strcpy(ptr, "overflow...");
```

In the above code, pointers to node and node.str are indistinguishable as they have the same address. Hence *node.str* has the bounds of the whole node object. When ptr is passed to *strcpy()*—even if *strcpy()* is instrumented—an overflow of *node.str* can overwrite the function pointer, potentially resulting in a serious bug or security vulnerability.

2.2 Pointer-based Approaches

An alternative approach is the pointer-based approach that 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 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 problem with object-table 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 [4], CCured [23], Cyclone [18], MSCC [34], and others [12, 24, 27] 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 [23] 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 no 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 for 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 avoid introducing inefficient WILD pointers and handle the memory layout incompatibility introduced by CCured use of fat pointers.

The most significant disadvantage of these pointer-based approaches 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 [23]. 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 [23, 34]. However, this can result in mirroring entire data structures. 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 [34] optimized encoding loses the ability to detect sub-object overflows.

2.3 Comparison of various approaches

Object-based and pointer-based approaches have complementary strengths and weaknesses. Object-base approaches are highly compatible because they do not change the memory layout. In fact, they have been successfully applied to the entire Linux kernel [11].

Scheme	No src code change	Complete (subfield access)	Memory layout	Arb. casts	Dyn. link lib
SafeC [4]	Yes	Yes	**No**	Yes	**No**
JKRLDA [13]	Yes	**No**	Yes	Yes	Yes
CCured-*Safe/Seq* [23]	**No**	Yes	**No**	**No**	**No**
CCured - *Wild* [23]	Yes	Yes	**No**	Yes	**No**
MSCC [34]	Yes	**No**	Yes	**No**	Yes
SoftBound	Yes	Yes	Yes	Yes	Yes

Table 1. Comparison of object-based (JKRLDA) and pointer-based approaches (CCured, SafeC, MSCC) in contrast to SoftBound with respect to attributes such as completeness, support for arbitrary casts, memory layout compatibility, support for dynamically linked libraries, completeness with respect to subfield accesses.

But, object-based approaches don't 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, as 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 [13, 23], 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 JKRLDA [13] satisfy most of the attributes except for completeness. CCured with only *Safe/Seq* pointers has low overhead and is complete but lacks source code compatibility. MSCC [34], has many positive qualities, but it is incapable of handling wild casts and it does not detect sub-object overflows in the configuration with the best performance 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 completeness of pointer-based schemes.

3. The SoftBound Approach

SoftBound is a compile-time transformation for inserting runtime bounds checks to enforce complete spatial safety of C programs. SoftBound is highly compatible with existing C source code because its disjoint metadata representation allows arbitrary casts and avoids memory layout change. 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 source code 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. 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 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 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) `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 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 = &array[100];
```

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. Thus, 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.

Shrinking Pointer Bounds 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). In such cases, the pointer inherits the bounds of the field it points to rather than the bounds of the entire structure:

```
typedef struct { ..., char str[8]; ...} node;
ptr = &node.str[2];
ptr_base = &node.str[0];
ptr_bound = &node.str[8];
```

This ability to shrink bound provides SoftBound with the capability to prevent internal object overflows. Further, SoftBound allows the programmer to explicitly shrink bounds using the `setbound()` function (*e.g.*, when employing a custom memory allocator).

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);
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;
table_lookup(ptr)->base = newptr_base;
table_lookup(ptr)->bound = newptr_bound;
```

Only load and stores of pointers are annotated; 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, the just described 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 transforms every function declaration and 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 unique identifier, specifying this function has been transformed by SoftBound:

```
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). Note that variable argument functions must be treated specially, as described in Section 5.

3.4 Differences & Similarities to CCured's WILD Pointers

In many respects, SoftBound's pointer representation is a more compatible and more efficient implementation of CCured's WILD pointers. Like CCured's WILD pointers, SoftBound provides complete 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 [23], 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.

Although complete, 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 protect 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 [23].

SoftBound's pointer representation improves upon WILD pointers while maintaining their spatial safety properties. First, Soft-

Bound'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 the 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 complete 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

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 [10].[2]

Due to the size and complexity of the full SoftBound implementation, we concentrate our efforts on an interesting 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 full SoftBound implementation includes mechanisms to deal with function pointers and sub-object bounds violations, the formalization of which we leave to future work.

At a high level we first develop a non-standard operational semantics for (straight-line) C programs that tracks information about which memory addresses have been allocated. Crucially, this semantics is *partial*—it is undefined whenever a bad C program would cause a (non sub-object) 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 that captures invariants ensured by a combination of the C compiler (e.g. that local variables map to stack addresses) and SoftBound instrumentation. Preservation and progress results then establish that, starting from a well-formed initial program, 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 that used in CCured's formalism [23].

4.1 Syntax

The grammar below 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 results of an arithmetic operation, a *lhs*, the address of a *lhs*, the results of a cast, the size of a type, or the results 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.

[2] A full description of the proof and accompanying Coq source code is available at `http://www.cis.upenn.edu/~jianzhou/softbound`

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

Table 2. Memory operations.

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

4.2 Operational Semantics

The operational semantics for this C fragment relies on an environment E, that has two components:

- A map, S, from variable names to their addresses and atomic types. (This models a stack frame.)
- A partial map, M, from addresses to values. (This models 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 doesn't affect any other location." The most interesting axioms have to do with `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. These operations are summarized in Table 2.

Given these operations, it is straightforward to define an operational semantics for this fragment of C that is undefined for programs that access unallocated memory locations. To model SoftBound's behavior, we augment this operational semantics to keep track of metadata. Values, written $v_{(b,e)}$ include their base (b) and bound (e) information along with the underlying data v. To fully capture the instrumentation performed by SoftBound it is also necessary for the operational semantics to propagate some type information as well—the types are used only in calculating metadata needed for assertions and otherwise do not affect the "real" underlying computation.

These considerations lead to three large-step evaluation rules. Left-hand-sides evaluate to a result r (which must be an address) and its atomic type a. They have no effect on the environment: $(E, lhs) \Rightarrow_l r : a$. A right-hand-side expression also yields a typed result, but it may also modify the environment E, causing it to become E': $(E, rhs) \Rightarrow_r (r : a, E')$. Commands simply evaluate to a result and final environment: $(E, c) \Rightarrow_c (r, E')$. In all three cases, r ranges over computational results, which include (metadata anno-

tated) values, `Abort`, `OutOfMem`, and `OK` (which is the the result of a successful command).

Space precludes showing the full set of operational rules (most of which are completely standard or obvious). We show only two examples that show how to evaluate a pointer dereference when the bounds check succeeds (left) and fails (right):

$$\frac{(E,lhs) \Rightarrow_l l : a* \quad \texttt{read}\ (E.M)\ l = \texttt{some}\ v_{(b,e)} \quad b \le v \wedge v + \text{sizeof}(a) \le e}{(E, *lhs) \Rightarrow_l v : a} \qquad \frac{(E,lhs) \Rightarrow_l l : a* \quad \texttt{read}\ (E.M)\ l = \texttt{some}\ v_{(b,e)} \quad \neg(b \le v \wedge v + \text{sizeof}(a) \le e)}{(E, *lhs) \Rightarrow_l \texttt{Abort} : a}$$

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 the well-formed type information, and a well-formed memory. A memory M is well formed when the metadata associated with each allocated location is well formed:

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

$$M \vdash_D d_{(b,e)} \triangleq \quad (b = 0) \vee [(b \neq 0) \wedge (\forall i \in [b,e).\ \texttt{val}\ M\ i) \wedge (\mathit{minAddr} \le b \le e < \mathit{maxAddr})]$$

Here, `val` M i is a predicate that is true when location i is allocated in memory M. $\vdash_M M$ guarantees that if any address is accessible, its value is in-bounds 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 type assigned by S.

With the above well-formed environment, 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* $\exists\ E'.\ (E,c) \Rightarrow_c (\text{OutofMem},E')$ *or* $\exists\ E'.\ (E,c) \Rightarrow_c (\text{Abort},E')$.

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 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()`, variable argument functions, function pointers, and arbitrary casts).

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). Such a mapping can be implemented in several ways including lookup trees or tables. SoftBound uses table lookup to implement this mapping.

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 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 bit selection 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, 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 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 Consideration

Handling arbitrary C programs requires handling 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 [13, 23], SoftBound easily supports separate compilation. As described ealier, SoftBound transforms functions to take additional parameters and changes the name of the function to signify that it has been transformed. Separate compilation works naturally, as the static or dynamic linker matches up caller and callee as usual. SoftBound's 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 (spatial safe) and untransformed (unsafe) verions of each function. For libraries that have not (yet) been transformed by SoftBound, library function wrappers similar to those used in MSCC [34] or CCured [23] (but without the marshaling issues caused by incompatible memory layout) may be employed.

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 handle the corresponding metadata. By default, SoftBound takes the safe (but slow) approach to `memcpy` (*i.e.*, the approach always inserts code to copy the metadata). Yet, most calls to `memcpy` involve buffers of non-pointer values. To address this inefficiency, SoftBound includes the option to infer 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 works well in practice to identify uses of `memcpy` involving pointers.

Variable argument functions To handle variable argument functions, SoftBound adds two extra parameters to all vararg functions. These parameters specify 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. SoftBound then performs checking in the `va_start` and `va_arg` macros to check that neither too many arguments nor too many pointer arguments are decoded. This is a change to the calling convention for vararg functions, but as SoftBound can also be used to transform library code, the change is hidden from the programmer.

Function pointers To protect function pointers, SoftBound 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 uses it to check when calling a function via a function pointer. Although this prevents data pointers or non-pointer data from being interpreted as a function pointer, cast between function pointers of incompatibly types presents a challenge because calling a function with arbitrary values may allow the manufacture of improper base and bounds. Although not yet implemented in our prototype implementation, the ultimate solution is to 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 benchmarks we have examined, such casts can be supported by allowing the programmer insert a call the `setbound()` function, which explicitly sets the bound for a pointer (including "unbounding" the pointer to allow it to access arbitrary memory if the programmer so desires).

Arbitrary casts and unions C supports arbitrary type conversion by explicit casts and implict 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 [4, 23, 34] have difficulty supporting arbitrary casts. In SoftBound, casts among pointer types simply inherit the same bounds. As described earlier, SoftBound enforces spatial memory safety and not specifically type safety. That is, SoftBound ensures that a variable can only dereference memory within its bounds; it does not provide an assurance about the types of those memory locations.

Memory reuse and stale metadata Although clearing metadata is not strictly necessary to preserve our well formed memory property, SoftBound clears the metadata when reallocating previously deallocated memory. To avoid clearing already zero metadata, SoftBound uses the heuristic that it clears only variables that likely had pointer metadata set. For variables on the stack, this entails setting the metadata to zero for all pointer variables or pointer fields in stack-allocated structs before the function returns. For variables in the heap, SoftBound examines the static type of the pointer being passed to `free()`. If the type is a pointer or struct with pointers, SoftBound inserts a code to clear the metadata before the call to `free()`.

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 [21] 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. By applying SoftBound post-optimization, it ensures SoftBound's instrumentation does not prevent code optimization. Furthermore, as register promotion and other optimizations have already reduced the number of memory operations, this reduces the amount of additional instrumentation introduced by SoftBound.

The SoftBound pass inserts code to (1) perform base/bound metadata manipulation prior to every memory operation that reads or writes a pointer, (2) perform a bounds check before every memory operation, (3) create a base and bound value for each pointer non-memory value in the program, and (4) rewrite all function calls to pass the base and bounds as was described in Section 3. Calls to external functions are mapped to wrapper functions as described in Section 3. These transformations are all strictly local (intra-procedural) transformations, without using any whole-program type inference or alias analysis.

The code for maintaining the base/bound metadata and performing the bounds check are written in standard C. The SoftBound pass invokes these routines by inserting function calls to these functions which are later inlined by subsequent LLVM passes.

After the intermediate code has been instrumented with SoftBound, we re-run the full suite of LLVM optimizations on the instrumented code. This simplifies the SoftBound pass, because subsequent optimization passes will remove some redundant checks 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, 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 less than 5000 lines of C++ code.

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 [32] and to versions of programs with well-documented security violations [22]. 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 [32] 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 attack based on the technique adopted, location of the overflow, and the attack target which is used to change

Attack and Target	Detection	
	Full	Store
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.* [32]. SoftBound's detection ability with full checking and store-only checking

	Detection?			
Benchmark	Valgrind	MudFlap	Store	Full
go	no	no	no	yes
compress	no	yes	yes	yes
polymorph	yes	yes	yes	yes
gzip	yes	yes	yes	yes

Table 4. Benchmarks with overflows and detection efficacy of SoftBound, Valgrind and Mudflap.

the control flow. SoftBound detects and prevents all these attacks in both complete and store-only checking mode. Publicly available tools such as StackGuard, ProPolice, Libsafe and Libverify miss more than 50% of these test cases [32].

We also evaluated SoftBound's ability to detect spatial bugs using actual spatial errors from real programs obtained from the BugBench suite [22]: 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 two popular debugging tools (Valgrind [24] and GCC's Mudflap [15]). 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, we also evaluated Valgrind and GCC's MudFlap efficacy. These experiments show that SoftBound detects violations missed by Valgrind and Mudflap. For example, Valgrind does not detect overflows on the stack, leading to its failure to detect some of the bugs.

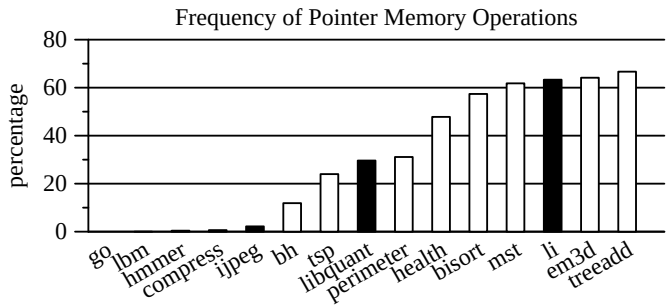

Figure 1. Percentage of the memory operations that load or store a pointer from/to memory, thus requiring a metadata access. The SPEC benchmarks are shaded dark.

6.3 Runtime Overhead Evaluation

Benchmarks We used 15 benchmarks selected from the SPEC CPU and the Olden benchmarks to evaluate SoftBound's performance. The Olden benchmarks [29] were used because they have been used in the most significant prior work in this area [13, 23, 34]. Our SoftBound prototype is not yet sufficiently robust enough to compile all of the C programs in SPEC 2006, but we present data for all of the programs we examined on which SoftBound works correctly. All runs are performed on a 2.66 Ghz Intel Core 2 processor. No modifications were made to any of the benchmarks, as discussed further in Section 6.4.

Benchmark characterization One of the main sources of overhead in SoftBound are the 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 1 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 negligible number of metadata accesses (less than 5%), including five of the seven 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 2 presents the percentage of runtime overhead of SoftBound over an uninstrumented baseline for the two metadata implementations with both full and store-only checking. The two left-most bars in each group of bars correspond to the overhead of complete checking using the hash table and shadow space implementations of the metadata facility. The benchmarks on the left of the graph (those with a lower frequency of metadata accesses) generally have lower overhead. On those benchmarks the overhead is largely performing the bounds checks, so the overhead is largely independent of the specific metadata encoding scheme. The overhead of these benchmarks could be reduced by applying more sophisticated bounds check removal techniques (*e.g.*, [7, 33]), which are complementary to and not the focus of this work.

Conversely, the runtime of the benchmarks on the right are significantly impacted by the metadata encoding, which is the focus on this work. The hash table encoding has the highest overhead, 127% on average. In most cases the larger overhead is due to the larger number of instructions necessary to perform the tag check. In a few benchmarks (*e.g.*, treeadd, mst, health), simulations of cache miss rates (not shown) indicate the additional memory pressure is contributing to the runtime overheads. The tag-less shadowspace

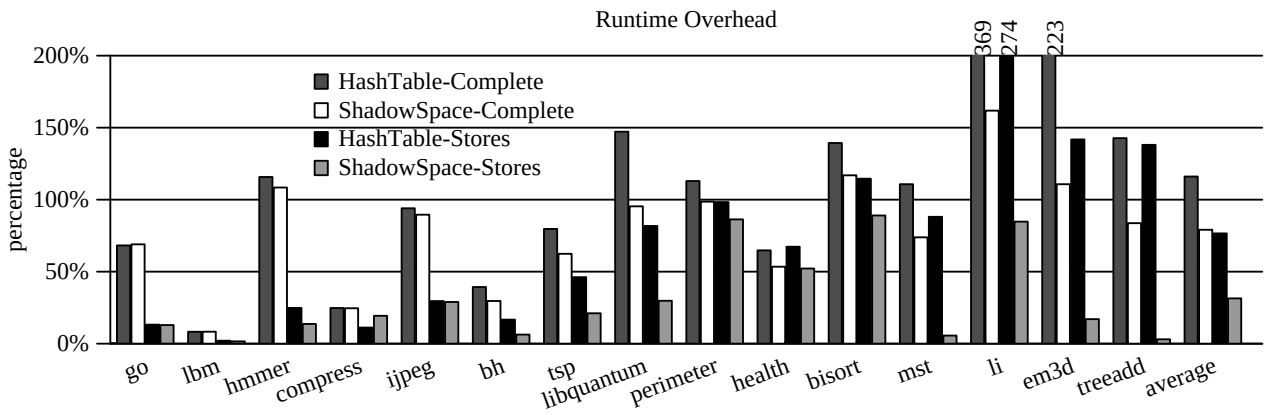

Figure 2. Normalized execution time overhead of SoftBound with full checking and store-only checking with two metadata organizations.

reduces this overhead to 79% on average. For all the benchmarks these runtimes overheads are likely more than acceptable for debugging, internal and external testing, and for mission-critical applications. Furthermore, the variation in overhead is significant. For example, if three benchmarks with the highest overhead (*i.e.*, li, bisort, em3d) are removed, the average overhead drops to 66%.

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 in that they are harder to diagnose because the manifestation of the bug is often widely separated from the root cause location at which the memory corruption occurred. The two right-most bars in each group of bars in Figure 2 shows the runtime overhead for the two metadata representations with store-only checking. The runtime overhead of checking only stores is reduced to just 32% on average. Furthermore, the runtime overhead of store-only checking is less than 15% for more than half of the benchmarks, which is likely low enough for production code.

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 fifteen benchmarks used in the performance evaluation, again, without any source modifications. In total, these benchmarks and network servers are approximately 130K total lines of code, all of which were transformed without modification, further supporting SoftBound's source code compatibility claim.

6.5 Performance Comparison to Related Approaches

CCured [23] and MSSC [34] are two pointer-based schemes closely related to SoftBound. CCured has low runtime overheads, ranging from 3% to 87% [23]. 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. More importantly, SoftBound supports separate compilation and requires no source code modifications. In contrast, 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 [23]. Lu *et al.* have 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 [22].

MSCC [34] 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 range from 17% to 185% with an average overhead of 68% [34]. Our own experimentation with MSCC and the published results shows that SoftBound's overhead is lower than MSCC's overhead on common benchmarks. For example, the SPEC benchmark `go` has 144% percent overhead with MSCC whereas SoftBound has 55% overhead. Moreover, MSCC does not handle arbitrary casts and in its best performing configuration has the same issues as object-based approaches in that it cannot prevent sub-object overflows.

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 [6, 14] and integer programming [16]. Static analysis has also been coupled with lightweight programmer or inferred annotations (*e.g.*, [17, 9]). 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 enforce control flow integrity [20] or dataflow integrity based on reaching definition analysis calculated statically [8]. Pointer analysis can also be used to compute the approximate set of objects written by each instruction [3]. In all three cases, these properties are checked dynamically, but neither strategy directly enforces memory safety. Probabilistic memory safety approaches, such as DieHard [5, 25], prevent 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 complete spatial safety for the C programming language. SoftBound provides completeness with no 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 complete spatial safety.

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 130k lines of code total) with no source code modifications. SoftBound's performance overhead is 79% and 31% 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 which is likely low enough to be employed in production code, substantially improving the security and robustness of real-world software systems.

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] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008.

[4] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *PLDI*, June 1994.

[5] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI*, June 2006.

[6] 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 *PLDI*, June 2003.

[7] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *PLDI*, June 2000.

[8] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[9] 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.

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

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

[12] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *ASPLOS*, Mar. 2008.

[13] 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.

[14] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. *SIGPLAN Not.*, 38(5):155–167, 2003.

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

[16] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[17] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, 2006.

[18] 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.

[19] 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.

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

[21] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[22] 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.

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

[24] 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.

[25] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

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

[27] 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.

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

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

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

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

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

[33] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination for the Java HotSpot client compiler. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, 2007.

[34] 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.

[35] 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.