View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Everything You Want to Know About Pointer-Based Checking

Santosh Nagarakatte[1], Milo M. K. Martin*[2], and Steve Zdancewic[3]

1 Department of Computer Science, Rutgers University, US
santosh.nagarakatte@cs.rutgers.edu
2 Computer and Information Sciences, University of Pennsylvania, US
milom@cis.upenn.edu
3 Computer and Information Sciences, University of Pennsylvania, US
stevez@cis.upenn.edu

Abstract

Lack of memory safety in C/C++ has resulted in numerous security vulnerabilities and serious bugs in large software systems. This paper highlights the challenges in enforcing memory safety for C/C++ programs and progress made as part of the SoftBoundCETS project. We have been exploring memory safety enforcement at various levels – in hardware, in the compiler, and as a hardware-compiler hybrid – in this project. Our research has identified that maintaining metadata with pointers in a disjoint metadata space and performing bounds and use-after-free checking can provide comprehensive memory safety. We describe the rationale behind the design decisions and its ramifications on various dimensions, our experience with the various variants that we explored in this project, and the lessons learned in the process. We also describe and analyze the forthcoming Intel Memory Protection Extensions (MPX) that provides hardware acceleration for disjoint metadata and pointer checking in mainstream hardware, which is expected to be available later this year.

1998 ACM Subject Classification D.3.4 Processors, D.2.5 Testing and Debugging

Keywords and phrases Memory safety, Buffer overflows, Dangling pointers, Pointer-based checking, SoftBoundCETS

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.190

1 Introduction

Languages like C and its variants are the gold standard for implementing a wide range of software systems from low-level system/infrastructure software to performance-critical software of all kinds. Features such as low-level control over memory layout, explicit manual memory management, and proximity to the hardware layout have made C the dominant language for many domains. However, C language implementations with their focus on performance do not ensure that programmers use C's low level features correctly and safely. Further, weak typing in C necessitates dynamic checking to enforce C language abstractions, which can cause additional performance overheads.

Memory safety ensures that all memory accesses are well-defined according to the language specification. Memory safety violations in C arise when accesses are to memory locations (1) that are beyond the allocated region for an object or an array (known as *spatial memory safety violations* or bounds errors) and/or (2) that have been deallocated while managing

* On leave at Google.

© Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic;
licensed under Creative Commons License CC-BY
1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 190–208

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

memory through manual memory management (known as *temporal memory safety violations*, use-after-free errors or dangling pointer errors).

Without memory safety, seemingly benign program bugs anywhere in the code base can cause silent memory corruption, difficult-to-diagnose crashes, and incorrect results. Worse yet, lack of memory safety is the root cause of multitude of security vulnerabilities, which result by exploiting a memory safety error with a suitably crafted input. The buffer overflow vulnerabilities, use-after-free vulnerabilities, and other low-level vulnerabilities resulting from memory safety violations have compromised the security of the computing ecosystem as a whole [42, 50, 8, 43, 3].

1.1 Memory Safety for C versus Java/C#

Languages such as Java and C# enforce memory safety with a combination of strong typing, runtime checks, and automatic memory management. The type casts are either disallowed or restricted. The runtime checks are performed to ensure that the array accesses are within bounds and type casts are between objects in the same object hierarchy. Automatic memory management (*e.g.*, using garbage collection) ensures that any reachable object is not freed.

Unfortunately, C's weak typing allows arbitrary type casts and conflates arrays and pointers. Preventing memory safety violations therefore requires checks on every memory access, because it is difficult to distinguish between pointers that point to a single element from pointers that point to an array of elements. Information about allocation/object sizes is lost in the presence of unsafe type casts. Hence, Java/C# style checking is infeasible to enforce memory safety for C. Further, avoiding temporal safety violations with automatic memory management (*e.g.*, using a garbage collector) does not allow the low-level control of memory allocations and deallocations to which C/C++ programmers are accustomed for implementing systems/infrastructure software.

As a consequence of C's weak typing and other low-level features, a pointer (*e.g.*, `void *p`) in C code can be (1) a pointer to a memory location allowed by the language specification (*e.g.*, arrays, structures, single element of a particular data type, and sub-fields in a structure), (2) an out-of-bounds pointer, (3) a dangling pointer pointing to deallocated memory locations, (4) a `NULL` pointer, (5) an uninitialized pointer, and (6) a pointer manufactured from an integer. To check whether a memory access (pointer dereference) is valid, the challenging task is not necessarily checking memory accesses; the primary challenge is maintaining and propagating sufficient information (metadata with each pointer) to perform such checking in the presence of C's weak typing and other low-level features.

1.2 State-of-the-Art in Enforcing Memory Safety for C

Given the importance of the problem, comprehensively detecting and protecting against memory safety violations is a well researched topic with numerous proposals over the years (see Szekeres *et al.* [48] for a survey). Lack of memory safety was originally regarded as a software quality problem. Thus, majority of techniques were debugging tools rather than *always-on* deployment of such solutions. Subsequently, when lack of memory safety resulted in numerous security vulnerabilities, many proposed solutions addressed the symptoms of these security vulnerabilities rather than the root cause of these errors (*i.e.*, lack of memory safety). The solutions that enforce memory safety can be broadly classified into three categories: tripwire approaches, object-based approaches, and pointer-based approaches.

Tripwire approaches place a guard block of invalid memory between memory objects. The guard block prevents contiguous overflows caused by walking past an array boundary with a

small stride. The tripwire approaches are generally implemented by tracking a few bits of state for each byte in memory; the additional bits indicate whether the location is currently valid [18, 40, 44, 49, 53]. When the memory is allocated, these bytes are marked as valid. Every load or store is instrumented to check the validity of the location. AddressSanitizer [46], a compiler implementation of the tripwire approach, is widely used to detect memory errors. Tripwire approaches can detect a class of buffer overflows (small strides) and use-after-free security vulnerabilities (when memory is not reused).

The object-based approaches [9, 12, 14, 22, 45] are based on the principle that all pointers are properly derived pointers to their intended referent (the object they point to). Hence these approaches check pointer manipulations to ensure that the resultant pointer points to a valid object. The distinguishing characteristic of this approach is that metadata 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* metadata. Object-based approaches keep the memory layout unchanged which increases the source compatibility with existing C code. Object-based approaches are generally incomplete in the presence of type casts between pointers and pointers to subfields of an aggregate data type. Further, they require mechanisms to handle out-of-bound pointers as creating out-of-bound pointers is allowed by the C standard. SAFECode [9, 12] and Baggy Bounds [1] are efficient implementations of the object-based approach using whole program analysis and allocation bounds, respectively.

The third approach is the pointer-based approach, which tracks metadata with each pointer. The metadata provides each pointer a view of memory that it can access according to the language specification. The pointer-based approach is typically implemented using a *fat pointer* representation that replaces some or all pointers with a multi-word pointer/metadata. Two distinct pointers can point to the same object and have different base and bound associated with them, so this approach overcomes the sub-object problem [5, 35, 47]. 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 with its metadata. Proposals such as SafeC [2], CCured [37, 7], Cyclone [21], MSCC [52], and others [11, 38, 41] use this pointer-based approach to provide varying degree of memory safety guarantees. Our SoftBoundCETS project was inspired by such prior pointer-based checking projects.

1.3 Goals of the Project

We started the SoftBoundCETS project[1] with the following goals:

- Comprehensive safety. Detect all memory safety errors in a fail stop fashion to completely prevent an entire class of memory safety related security vulnerabilities.
- Low performance overhead. Memory safety enforcement has to be carried out on deployed programs to prevent security vulnerabilities. Hence, any performance overhead should be indiscernible to the end user to be widely adopted.
- Source compatibility. As significant C code base exists, we wanted to enforce memory safety without requiring changes to existing code (*i.e.*, maintain source compatibility). However, recompilation of the code is expected.

The prior solutions that inspired our project failed to attain all of the above goals. Cyclone [21] was comprehensive with low overhead, but it required significant code modifica-

[1] The name SoftBoundCETS is a concatenation of the names of its components: SoftBound [35] and CETS [36].

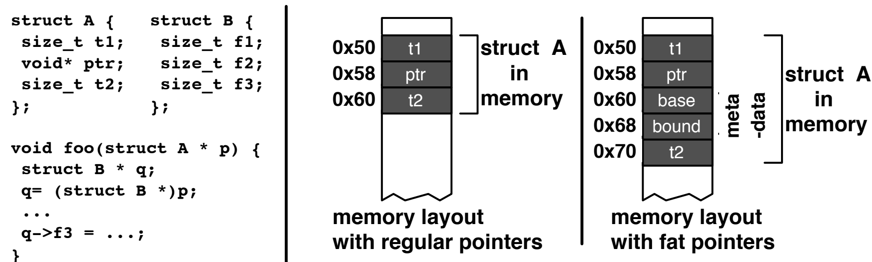


Figure 1 Memory layout changes in a program with fat pointers. The code snippet also shows how a pointer involved in arbitrary type casts can overwrite the pointer metadata.

tions. CCured attained the goals of reasonable performance overhead and comprehensiveness for providing spatial safety. CCured maintained metadata with pointers by changing the representation of a pointer into a fat pointer (ptr, base, bound). It relied on a garbage collector to provide temporal safety.

However, there were two major drawbacks with the CCured approach: interfacing with libraries and type casts. With the use of a fat-pointer, CCured required marshalling/demarshalling of pointers, which resulted in deep copies of data structures while interfacing with external libraries through wrappers. In the presence of unsafe type casts, the pointer metadata could be potentially overwritten (see Figure 1). CCured used a whole program inference to detect such pointers involved in casts (*e.g.*, WILD pointers), which prevented separate compilation and WILD pointers had higher performance overhead. To mitigate these issues, the programs had to be changed/rewritten to avoid such type casts or use run-time type information (RTTI) extensions.

We started our project to address compatibility issues with CCured in an effort to make it easily usable with large code bases while avoiding garbage collection. Our goal was to provide checked manual memory management in contrast to automatic memory management. The SoftBoundCETS project enforces memory safety by injecting code to maintain per-pointer metadata and checking the metadata before dereferencing a pointer. To provide compatibility, the per-pointer metadata is maintained in a disjoint metadata space leaving the memory layout of the program unchanged. We have implemented pointer-based checking in various ways (see Table 1): within the compiler, in hardware, and with hardware instructions for compiler instrumentation. We have used the compiler instrumentation prototype of SoftBoundCETS to compile more than a million lines of C code. The latest compiler prototype is available at `https://www.cs.rutgers.edu/~santosh.nagarakatte/softbound/`. Intel has recently announced Memory Protection Extensions [20], which provides hardware acceleration for a similar pointer-based compiler instrumentation for enforcing spatial safety.

In the next section, we describe our approach, design decisions, various implementations and their trade offs. In Section 3, we describe the Intel Memory Protection Extensions (MPX) [20], similarities/differences between SoftBoundCETS and MPX while highlighting the pros and cons of the design decisions. We reflect on the lessons learned in Section 4.

2 Pointer-Based Checking with Disjoint Metadata

The SoftBoundCETS project uses a pointer-based approach, which maintains metadata with each pointer. The metadata provides the pointer a view of the memory that it can safely access. Although pointer-based checking has been proposed and investigated earlier [37, 2, 21, 52], the key difference is that our approach maintains the metadata disjointly in a shadow memory

Table 1 Various implementations of pointer-based checking developed as part of the SoftBound-CETS project, distinguished based on instrumentation method (Instr.), support for spatial safety, temporal safety, instrumentation of integer operations, support for check optimizations, performance overhead, and data structures used for disjoint metadata.

	Instr.	Spatial safety	Temporal safety	Instr. integer ops	Check Opts	Slow -down	Disjoint meta-data
HardBound [11]	Hardware	Yes	No	Yes	No	5-20%	Shadow
SoftBound [35]	Compiler	Yes	No	No	Yes	50-60%	Hash/ Shadow
CETS [36]	Compiler	No	Yes	No	Yes	30-40%	Trie
SoftBound-CETS [31]	Compiler	Yes	Yes	No	Yes	70-80%	Trie
Watchdog [32, 33]	Hardware	Yes/No	Yes	No	No	10–25%	Shadow
WatchdogLite [34]	Hybrid	Yes	Yes	No	Yes	10–20%	Shadow

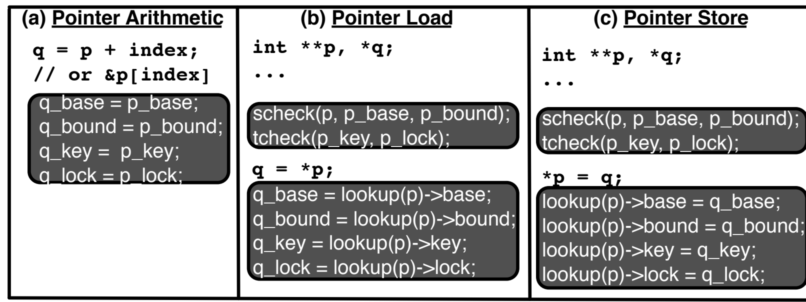


Figure 2 (a) Pointer metadata propagation with pointer arithmetic, (b) metadata propagation through memory with metadata lookups on loads, and (c) metadata lookups with pointer stores.

region, which provides an opportunity to revisit pointer-based checking (generally considered invasive) for retrofitting C with practical memory safety satisfying the above three goals.

We describe the main design choices with our approach: (1) metadata for spatial safety, (2) metadata for temporal safety, (3) propagation of metadata, and (4) organization of the metadata shadow space. We also describe how it enforces comprehensive detection in the presence of type casts and provides compatibility with existing C code while supporting external libraries. Figure 2 and Figure 3 illustrate the pointer-based metadata, propagation and checking abstractly using pseudo C code notation. We use the term SoftBoundCETS interchangeably to refer to both our approach and the various prototypes built in this project shown in Table 1.

2.1 Spatial Safety Metadata

To enforce spatial safety, the base and bound of the region of memory accessible via the pointer is associated with the pointer when it is created. The base and bound are each typically 64-bit values (on a 64-bit machine) to encode arbitrary byte-granularity bounds information. These per-pointer base and bounds metadata fields are sufficient to perform a bounds check prior to a memory access (Figure 3d). This representation permits the creation of out-of-bounds pointers and pointers to the internal elements of objects/structs and arrays (both of which are allowed in C/C++).

2.2 Temporal Safety Metadata

To enforce temporal safety, a unique identifier is associated with each memory allocation (Figure 3a). Each allocation is given a unique 64-bit identifier and these identifiers are never reused. To ensure that this unique identifier persists even after the object's memory has been deallocated, the identifier is associated with all pointers to the object. On a pointer dereference, the system checks that the unique allocation identifier associated with the pointer is still valid.

Performing a validity check on each memory access using a hash table or a splay tree can be expensive [2, 22], so an alternative is to pair each pointer with two pieces of metadata: an allocation identifier – the *key* – and a *lock* that points to a location in memory called *lock location* [41, 52, 36, 5, 32]. The key and value at the lock location will match if and only if the underlying memory for the object is still valid (*i.e.*, it has not been deallocated). Rather than a hash table lookup, a dereference check then becomes a direct lookup operation – a simple load from the lock location and a comparison with the key (Figure 3c). Freeing an allocated region changes the value at the lock location, thereby invalidating any other (now-dangling) pointers to the region (Figure 3b). Because the keys are unique, a lock location itself can be reused after the space it guards is deallocated.

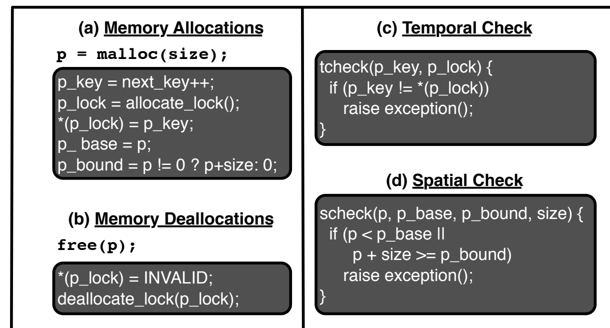


Figure 3 (a) Pointer metadata creation on memory allocations, (b) identifier metadata being invalidated on memory deallocations, (c) lock and key checking using identifier metadata, and (d) spatial check performed using bounds metadata.

2.3 Metadata Propagation

The metadata – base, bound, lock, and key – are associated with a pointer whenever a pointer is created. These metadata are propagated on pointer manipulation operations such as copying a pointer or pointer arithmetic (Figure 2a). The metadata for pointers in memory is maintained in a disjoint metadata space [11, 35, 32, 36, 17]. Figure 4 illustrates the metadata maintained in the disjoint metadata space. The disjoint metadata space protects the metadata from malicious corruption and leaves the memory layout of the program intact, retaining compatibility with existing code.

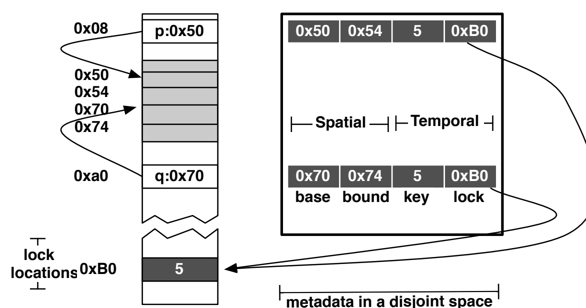


Figure 4 Metadata maintained with each pointer in memory with SoftBoundCETS. There are two pointers **p** and **q** which point to different sub-fields in the same allocation. Hence they have different bounds metadata but the same lock and key metadata.

The metadata is propagated with pointer arguments across function calls. If the pointer

arguments are passed and returned on the stack, metadata for these pointer arguments are available through accesses to the disjoint metadata space. However, in practice propagating metadata with function calls is not straightforward. Arguments are often passed in registers according to the function calling conventions of most ISAs, and C allows variable argument functions. Furthermore, function calls (indirect calls) can be made through function pointers. Function pointers can be created through unsafe type casts to functions with incompatible types. Calling a function through such a function pointer should not be allowed to manufacture metadata, which may result in memory accesses to arbitrary memory locations.

We have explored two approaches to propagate metadata for pointer arguments: adding metadata as extra arguments [35, 36] and using a shadow stack for propagating metadata [31]. We pass metadata as extra arguments for functions that are not involved in type casts and used with indirect calls. We use a shadow stack for all other function calls including variable argument functions. The shadow stack provides a mechanism for dynamic typing between the arguments pushed at the call site and the arguments retrieved by the callee. The shadow stack is slower but it ensures that the callee never successfully dereferences a non-pointer value pushed by the caller in the call stack by treating it as a pointer value. An exception is triggered only when such pointers are dereferenced but not when they are created in accordance with the C specification.

2.4 Implications of Disjoint Metadata Design

To use disjoint metadata, we had to address the following questions: (1) How are memory locations mapped to their disjoint metadata? (2) Is metadata maintained for every memory location or only for memory locations with pointers?, and (3) Is metadata updated on every memory access?

Different implementations of the disjoint metadata space have different memory overhead and performance trade-offs. We have explored three implementations: shadow space (a linear region of memory) [11, 35, 32], a hash table [35], and a trie data structure [39, 36, 16]. Shadow space is beneficial when the size of the metadata is significantly smaller than the granularity of the memory region (*e.g.*, 1-bit of metadata for every 16 bytes). Hash tables can experience conflicts, and resizing the hash table causes overheads. In th end, we settled upon a two level trie data structure that maps the entire 64-bit virtual address space. Although mapping exists for the entire virtual address space, the entries in the trie are allocated only when the metadata is used.

Disjoint metadata accesses are expensive compared to fat pointers because additional instructions are required to translate a memory address to the corresponding metadata address. To reduce performance overheads, SoftBoundCETS maintains metadata in the disjoint metadata space only for pointers in memory and performs metadata loads (stores) only when the loaded (stored) value from (into) a memory location is a pointer as shown in Figure 2b and Figure 2c, respectively. The metadata for pointers in temporaries (registers) are maintained in temporaries (registers). Hence, the accesses to the disjoint metadata space typically occur with pointer-chasing code or linked data structures. Most programs have fewer pointers in memory compared to data and metadata is maintained only with pointers. Hence, the memory overhead is significantly lower than the worst-case 4× overhead (about 50% on average for SPEC benchmarks).

SoftBoundCETS initializes metadata to an invalid value when a pointer is created from an integer (in registers), which causes any subsequent check on such pointer dereferences to raise an exception. SoftBoundCETS does not access the disjoint metadata space when non-pointer values are written (read) to (from) memory. However, a side effect of instrumenting only

pointer operations is that a pointer can be manufactured from a integer through memory. However, SoftBoundCETS allows such a dereference through a manufactured pointer only when the resulting pointer belongs to the same allocation and is within bounds of the object pointed by the pointer before the cast. We illustrate how SoftBoundCETS still provides comprehensive protection against memory errors with such type casts below.

2.5 Comprehensive Protection in the Presence of Type Casts

Our approach enforces comprehensive memory safety because (1) metadata is manipulated/accessed only through the extra instrumentation added, (2) metadata is not corrupted and accurately depicts the region of memory that a pointer can legally access, and (3) all memory accesses are conceptually checked before a dereference.

Unlike fat pointers, a store operation using a pointer involved in an unsafe type cast can only overwrite pointer values but not the metadata in the disjoint metadata space. When pointers involved in arbitrary casts are subsequently dereferenced, the pointer is checked with respect to its metadata. As the correctness checking uses the metadata to ascertain the validity of the memory access and the metadata is never corrupted, our approach ensures comprehensive detection of memory safety errors.

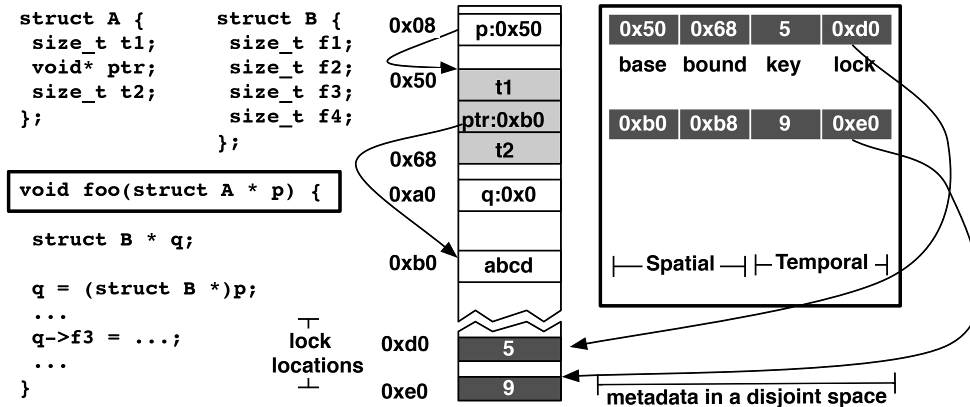
Figure 5 pictorially represents the disjoint metadata space as it is updated in the presence of arbitrary type casts. We will refer to this example later to illustrate a subtle interplay between type casts and comprehensive protection with Intel MPX. Figure 5(a) shows a program with two structure types `struct A` and `struct B` and a function `foo`, which has pointer `p` of type `struct A` as an argument. The location pointed by pointer `p` is allocated and resident in memory as shown in Figure 5(a). The sub-field `ptr` in the allocated memory region pointed by pointer `p` points to some valid memory location. Pointers `p` and `ptr` are resident in memory and have metadata in the disjoint metadata space as shown in Figure 5(a). Figure 5(b) depicts the execution of the program where it creates pointer `q` from pointer `p` through an arbitrary type cast. The metadata for pointer `q`, which is assumed to be resident in memory, is copied from pointer `p`.

The program writes integers to memory locations using pointer `q` as shown in Figure 5(c). As a result, `ptr` sub-field is overwritten with arbitrary non-pointer values. Our approach does not access the disjoint metadata space on integer operations. Hence, the metadata is not corrupted. When the program later tries to dereference `ptr` in memory, the dereference would be checked with respect to its metadata and memory safety errors would be detected.

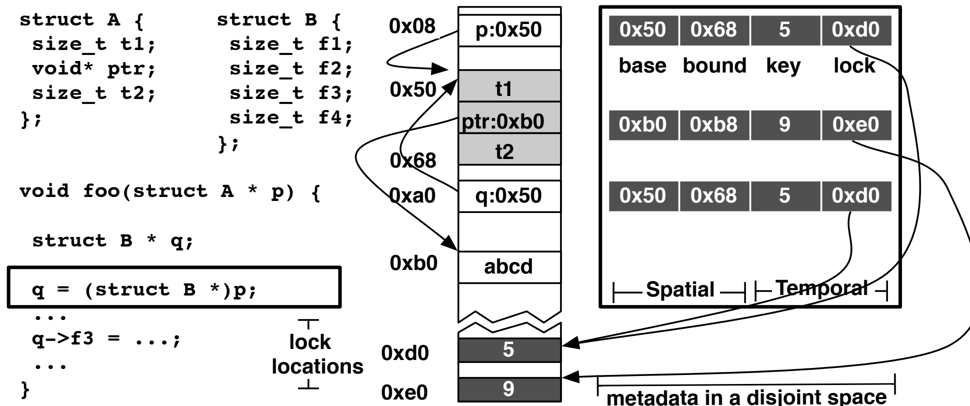
2.6 Compatibility with Existing Code and Libraries

To enforce memory safety with legacy programs, SoftBoundCETS handles programs with type casts, supports separate compilation, and provides wrappers for commonly used libraries. The use of disjoint metadata enables comprehensive protection with type casts. Rewriting C source code to avoid type casts is not necessary.

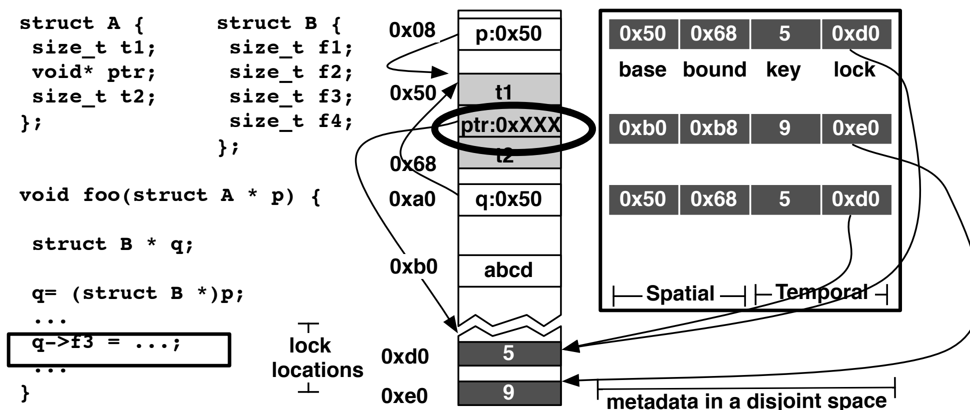
SoftBoundCETS supports separate compilation because the instrumentation is local and does not require whole program analysis in contrast to CCured [37]. Separate compilation also allows creation of memory-safe libraries. In the absence of library sources, code compiled with SoftBoundCETS can interface with library code through wrappers for the exported library functions. In the absence of such wrappers, code instrumented with SoftBoundCETS will not experience false violations as long as the external libraries do not return pointers or update pointers in memory.



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



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



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

Figure 5 This figure illustrates how disjoint metadata protects the metadata. Writes to memory locations involved in arbitrary type casts can only modify pointer values (*ptr* field in the *struct A*) but not the metadata. When the pointer *ptr* is dereferenced, the dereference will not be allowed and the memory safety violation would be caught.

When an external library returns or updates pointers in memory, wrappers provide the glue code between the instrumented code and the external library. Writing wrappers is easier with SoftBoundCETS compared to CCured [37] because it is not required to perform deep copies of data structures when memory-safe code interfaces with an external library. However, the disjoint metadata space should be updated in the wrapper whenever the external library updates pointers in memory. Although our approach makes it easier to write wrappers, it can be tedious and error prone for some libraries. We provide wrappers for the commonly used libraries (*e.g.*, Linux utilities, libc, and networking utilities) with our publicly available compiler prototype. Intel MPX further makes it even easier to incrementally deploy spatial safety checking by storing the pointer value redundantly in the metadata space to be permissive when non-instrumented code modifies the pointer and does not properly update the bounds metadata, which presents a compatibility/safety trade off as described in Section 3.

2.7 Efficient Instrumentation and Challenges

One of the key requirements for low performance overhead is efficient instrumentation and implementation of the pointer-based checking approach described above. Table 1 lists the numerous designs that we have explored in the hardware-software tool chain with different types of instrumentation, safety guarantees, and performance trade-offs [35, 36, 32, 33, 31, 10, 34]. Binary instrumentation and source-to-source translation have been popular with various proposals for enforcing partial memory safety. Source-to-source translation is widely used because pointer information is readily available in the source code. Binary-based techniques have been popular partly due to the availability of the dynamic binary translation tools like PIN [29] and Valgrind [40].

Our project benefited, albeit serendipitously, from the LLVM compiler that maintains pointer information from the source code in the intermediate representation (IR) [26]. Instrumenting within the compiler provides three main benefits: (1) checking can be performed on optimized code after executing an entire suite of conventional compiler optimizations, (2) pointers and memory allocations/deallocations can be identified precisely by leveraging the information available to the compiler, and (3) a large number of checks can be eliminated statically using check elimination optimizations.

Figure 6 presents the percentage runtime overhead of the latest compiler-based SoftBound-CETS prototype within the LLVM compiler (LLVM-3.5.0) over an un-instrumented baseline. SoftBoundCETS enforces comprehensive memory safety at 76% overhead on average with SPEC benchmarks. The overhead is significantly lower with I/O intensive benchmarks and various utilities such as OpenSSL, Coreutils, and networking utilities (less than 30%). A variant of pointer-based checking described above that propagates metadata with all pointer loads and stores but checks only store operations can enforce memory safety at 23% overhead on average with SPEC benchmarks. Store-only checking, which allows read operations on out-of-bound locations and with dangling pointers, is sufficient to prevent all memory corruption based security vulnerabilities.

The remaining performance overhead with the compiler instrumentation is attributed to the following sources: (1) spatial checks (5 x86 instructions), (2) temporal checks (3 x86 instructions), (3) metadata loads (approximately 14 x86 instructions), (4) metadata stores (approximately 16 x86 instructions), (5) shadow stack accesses, and (6) additional spills and restores due to register pressure.

To further reduce overheads with compiler-based approaches, we have explored memory safety instrumentation within hardware with varying amounts of hardware support [11, 32,

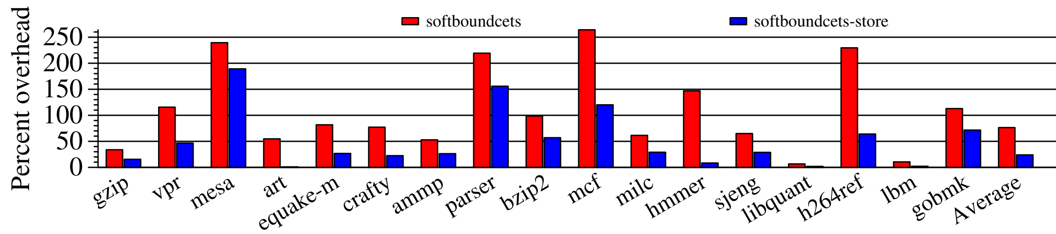


Figure 6 Runtime execution time overheads of the SoftBoundCETS compiler prototype with comprehensive memory safety checking (left bar of each stack) and while checking only stores (right bar of each stack) on a Intel Haswell machine. Smaller bars are better as they represent lower runtime overheads.

33]. HardBound [11] and Watchdog [32, 33] implicitly check every memory access within hardware by receiving information about pointer allocations from the runtime with additional instructions. The benefits of implicit checking within hardware are streamlined execution of checks and metadata accesses and the reduction in the number of register spills/restores by leveraging hardware registers for metadata.

In the final design point we proposed, hardware provides acceleration for spatial checks, temporal checks, metadata loads and metadata stores with ISA extensions and the compiler performs metadata propagation, performs check elimination, introduces check instructions and metadata access instructions into the binary [34]. This division of labor between the software stack and the hardware reduces the invasiveness of the hardware changes. With compiler-inserted checking instructions, we have demonstrated that comprehensive memory safety can be enforced at approximately 20% performance overhead on average with SPEC benchmarks. Further, these hardware instructions prevent all memory corruption vulnerabilities with store-only checking with approximately 10% performance overhead on average for SPEC benchmarks. Intel MPX has adopted a similar approach with compiler-based instrumentation with hardware instructions for enforcing spatial safety.

3 Intel Memory Protection Extensions

Intel has announced the specification of Memory Protection Extensions (MPX) [20] for providing hardware acceleration for compiler-based pointer-based checking with disjoint metadata slated to appear in the "Skylake" processors later in 2015. Like SoftBoundCETS with hardware instructions [34], MPX (1) provides hardware acceleration for compiler-based pointer-based checking, (2) uses disjoint metadata for the pointers in memory, and (3) provides ISA support for efficient bounds checking. Significantly, MPX does not address use-after-free errors. MPX uses the same basic approach pioneered by the SoftBoundCETS project, and this section describes some of the differences.

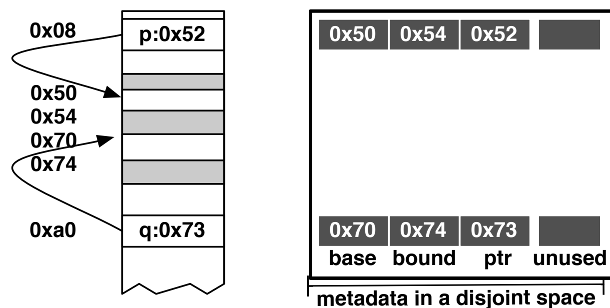


Figure 7 MPX maintains four pieces of metadata with each pointer: base, bound, the pointer redundantly in the metadata space and the fourth unused space.

3.1 MPX Instructions and Operation

The Intel MPX extension provides new instructions that a compiler can insert to accelerate disjoint per-pointer metadata accesses and bounds checking. MPX aims for seamless integration with legacy and MPX code with minimal changes to the source code. One design goal of MPX is to allow binaries instrumented with MPX to execute correctly (but without performing bound checks) on pre-MPX hardware. MPX achieves this goal by selecting opcodes that are NOOPs (no operation) on existing x86 hardware for the new instructions. Hence, a MPX binary can run on MPX-enabled machines and non-MPX machines.

MPX introduces four new 128-bit bound registers (B0-B3). MPX provides the following instructions: (1) `BNDCL` – check pointer with its lower bound, (2) `BNDCU` – check pointer to the upper bound, (3) `BNDSTX` – store metadata to the disjoint metadata space, (4) `BNDLDX` – load metadata from the disjoint metadata space, and (5) `BNDMK` – to create bounds metadata for a pointer. MPX also extends function calling conventions to include these bound registers.

A key innovation of MPX beyond our work is the support for incremental deployment of bounds checking. MPX redundantly stores the pointer value in the metadata space along with the base and bound metadata as shown in Figure 7. When a pointer is loaded from memory, the corresponding metadata is loaded from the disjoint metadata space and the loaded pointer is compared with the pointer redundantly stored in the metadata space. If the pointer in the metadata space and the pointer loaded do not match (typically occurs when the non-instrumented code modifies the pointer and does not properly update the bounds metadata), then MPX allows the pointer to access any memory location by un-bounding it. MPX design adopts the compatible-but-unsafe model, allowing best-effort checking of instrumented code until all code has been recompiled for MPX.

3.2 Type Casts and Comprehensiveness with Intel MPX

The Intel MPX's support for incremental deployment of bounds checking results in the loss of comprehensiveness in the presence of insidious type casts from integers to pointers either directly or indirectly through memory. Particularly, the arbitrary pointer manufactured through type casts in Figure 5(c) will be allowed by MPX to access any location in memory because (1) the pointer in the metadata space is not updated during an integer store, (2) the pointer loaded and the pointer in the metadata space would mismatch on a metadata load, and (3) the result is an un-bounded pointer. The compiler can identify the occurrence of such type casts either implicitly or explicitly and warn the programmer about them. Nevertheless, MPX is a step in the practical deployment of memory safety checking in production. Although Intel MPX may appear permissive, the spatial safety protection provided by Intel MPX is similar or stronger to the protection provided by fat-pointer approaches while easing the problem of interfacing with external libraries. Although the current ISA specification does not disable this behavior with un-bounding, Intel could easily add a stricter mode that changes this behavior to be safer by default.

4 Reflections on Memory Safety Enforcement

We briefly describe our experience in enforcing memory safety with large code bases and reflect on the performance overheads, various aspects of the language, implementation, and the hardware-software stack. We describe changes to the language and the hardware-software stack, which can potentially make memory safety enforcement inexpensive while being expressive for C's domain.

4.1 Performance Overheads

Low performance overhead is one of the key requirements for adoption of memory safety enforcement techniques. Our experiments indicate that compiler instrumentation with hardware acceleration in the form of new instructions can provide comprehensive memory safety with approximately 10–20% performance overhead for computation-based SPEC benchmarks and less than 10% I/O intensive utilities and applications. These overheads are within the reach of the acceptable threshold for a large class of applications. For example, a large fraction of C code is compiled at the `O2` optimization level because the code is unstable with higher optimization levels (likely due aggressive optimizations by the compiler in the presence of undefined behavior and implementation-dependent behavior [51]). For many "fast enough" applications, an additional 5–10% overhead for enforcing memory safety will not be perceivable to the user. The store-only instrumentation within the compiler with hardware acceleration can prevent all memory corruption based security vulnerabilities is an attractive option for reducing overheads further. Store-only checking provides much better safety than control-flow integrity with similar performance overheads [24, 15]. An application's observed performance overhead depends on the memory footprint, amount of pointer-chasing code, and the locality in the accesses. Hence, it remains to be seen what performance cost users will tolerate in exchange for the security and safety of the computing ecosystem.

We have explored only simple check optimizations with our prototype. A wide range of check optimizations based on loop invariant code motion and loop peeling can reduce overhead. For example, compiler optimizations based on weakest preconditions have reduced the performance overhead of spatial safety enforcement by 37% for the SPEC benchmarks [13]. Moreover, simple code transformations can significantly reduce performance overhead. For example, a small change to the data structure used by the SPEC benchmark `equake` not only improved the baseline execution time performance by 60% but also reduced overhead of memory safety with SoftBoundCETS from 3× to 30%.

4.2 Implementation Specific C Dialects

Other than performance, one of the biggest impediments in the adoption of memory safety enforcement techniques is the pervasiveness of implementation-specific C dialects. In contrast to the C standard, a fraction of the C code base depends on the behaviors provided by contemporary C implementations [4]. Are deviations from the C standard acceptable? There is no unanimous answer but such reliance on implementation specific behaviors result in non-portable code and can be exploited aggressively by an optimizing compiler (see classification of undefined behaviors in LLVM in [28]). We highlight examples that are arguably not in conformance with the C standard, which can cause false violations with our approach.

4.2.1 Narrowing of Bounds for Sub-Objects

When the program creates a pointer to a sub-field of an aggregate type, should the bounds of the resultant pointer be just the sub-field or the entire aggregate object? Our approach provides the ability to easily narrow the bounds of pointers, which in turn allows us to prevent internal object overflows. When instructed to check for overflows within an object, we shrink the bounds of pointer when creating a pointer to a field of a *struct* (*e.g.*, when passing a pointer to an element of a *struct* to a function). In such cases, we narrow the pointer's bounds to include only the individual field rather than the entire object.

Although most programmers expect shrinking of bounds, it can result in false violations for particularly pathological C idioms. 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* [23]. Casts do not narrow bounds, so one idiom that will not cause false violations is casting a pointer to a *struct* to a *char** or *void**.

Another case in which we do not narrow bounds is when when creating a pointer to an element of an array. Although tightening the bounds is such cases may often match the programmer's intent, C programs occasionally use array element pointers to denote a sub-interval of an array. For example, a program might use *memset* to zero only a portion of an array using *memset(&arr[4], 0, size)* or use the *sort* function to sort a sub-array using *sort(&arr[4], &arr[10])*. We have found these assumptions and heuristics match the source code we have experimented with, but programmer can control the exact behavior with compiler command line flags.

4.2.2 Integer Arithmetic with Integer to Pointer Casts

Masking pointer values is a common idiom in low-level systems code to use the lower order bits of the pointer representation to store additional information with aligned pointers. The last three bits are always zero in the pointer representation on a 64-bit machine with an aligned pointer. These bits can be used to store tags, which can maintain additional information about the pointer. Such operations would involve casting pointers to integers, masking the values, and casting integers back to pointers. Such casts may also be performed after arbitrary integer arithmetic. Occasionally, pointers can be created from integers due to (old) interfaces to utilities. For example, the interface to the system functions in Microsoft Windows on 32-bit systems (*Kernel32.dll*) used integers for pointers, which resulted in pointer to integer and integer to pointer casts. Our approach will set the metadata of the pointer cast from an integer to be invalid and raise an exception on dereferencing such pointers. Avoiding such exceptions require explicit setting of bounds by the programmer. If such cast operations occur through memory, our approach allows dereference of such pointers as along as the pointer type cast from an integer belongs to the same allocation and is within bounds of the object pointed by the pointer before the cast.

4.3 Interoperability and Engineering

Pointer-based checking is more invasive compared to other approaches as each pointer operation needs to be tracked, propagated with metadata, and checked. Hence, significant engineering is required to make it practical. AddressSanitizer [46], which uses a tripwire approach, is less invasive but still required significant engineering for use with code bases of the Chromium browser and other utilities. Our experience indicates that significant engineering and enabling memory safety checking using a simple compile time flag is crucial for adoption. Maintaining interoperability with un-instrumented code is necessary to encourage incremental deployment. Intel MPX with its support for incremental deployment will likely increase the adoption of memory safety enforcement with pointer-based checking.

4.4 Language Design Considerations

We provide some suggestions for the future standards of the C programming language to ease memory safety enforcement. One way to ease memory safety enforcement is by restricting type casts. Enforcing spatial safety with strong typing is easier compared to weak typing (similar to observations in CCured [37]). More restrictive type cast rules imply easier spatial safety enforcement because only array bounds need to be checked. Unfortunately, implementing large software systems (with some form of polymorphism/reusable code) requires type casts in C. Object oriented variants of C such as C++ have reasonable subsets that can enforce spatial safety while being suitable for building infrastructure/systems code [10].

Even when type casts are necessary, preventing the creation of pointers from non-pointers will enable easier memory safety enforcement for a large class of applications compared to the state-of-the-art now. When creation of pointers from non-pointers is essential, explicitly creating a separate data type other than integers will ease the enforcement of memory safety.

The C language conflates arrays and pointers, which requires bounds checking on every memory access. Annotations that identify non-array pointers, which point to a single element in contrast to an array of elements, can avoid such checking. In the case of linked data structures, such annotations will significantly reduce spatial safety overheads by reducing not only the overheads due to checks but also disjoint metadata accesses. The overhead with Intel MPX for spatial safety can be reduced significantly by revisiting annotations similar to Cyclone [21] and Deputy [6].

4.5 Implementation Considerations

We highlight the implementation considerations for efficient pointer-based checking with compiler assisted instrumentation and hardware acceleration similar to MPX.

Type information in the compiler is known to benefit numerous domain-specific instrumentation [30]. Maintaining information about types in the program within the compiler tool chain will ease the enforcement of memory safety especially with Intel MPX. If maintaining full type information is infeasible, the compiler should at least maintain information about pointer and non-pointer types. This information must be preserved and maintained with compiler optimizations, which would enable memory safety instrumentation on optimized code. The LLVM compiler maintains best effort type information, which helped the SoftBoundCETS project to perform pointer-based checking within the compiler with low performance overhead.

Automatic memory management through garbage collection is an attractive alternative to checked manual memory management. Automatic memory management can increase programmer productivity as they do not have to manually manage memory. However, garbage collection's effectiveness decreases in the presence of weak typing with C and can cause memory leaks. The pointer-based metadata can be used to perform precise garbage collection [10, 4]. In contrast to identifier metadata for enforcing temporal safety, spatial safety metadata can be used enforcing temporal safety by setting the bounds of the deallocated pointer and all its aliased pointers in the metadata space to a invalid value [47]. Any subsequent dereference of such a deallocated pointer will raise an exception as the bounds metadata is invalid. Similar dangling pointer nullification has been proposed to detect use-after-free errors [27]

4.6 Multithreading

A pointer-based approach should ensure atomicity of the checks, metadata accesses, and the memory access in multithreaded programs, which can be ensured by relying on the

compiler. If a pointer operation, temporal safety check and the metadata accesses occur non-atomically, interleaved execution and race conditions (either unintentional races or intentional races) can result in false violations and miss true violations. To avoid them, the compiler instrumentation must ensure that: (1) a pointer load/store's data and metadata access execute atomically, (2) checks execute atomically with the load/store operation, and (3) allocation of metadata is thread safe. The compiler can satisfy requirement #3 by using thread-local storage for the identifiers. The compiler can ensure requirements #1 and #2 for data-race free programs by inserting metadata access and check instructions within the same synchronization region as the pointer operation. For programs with data races, if the compiler can perform the metadata access as an atomic wide load/store and perform the temporal check after the memory operation, then the approach can detect all memory safety violations (but can experience false violations). Alternatively, the compiler can either introduce additional locking or use best-effort bounded transactional memory support in latest processors [19, 25](*e.g.*, Intel's TM support in Haswell) to avoid false violations.

5 Conclusion

Memory safety enforcement is the job of the language implementation. A language implementation that is compatible with the C standard can enforce comprehensive memory safety at low overheads with pointer-based checking. Restrictions on the creation of pointers from non-pointers, annotations distinguishing pointers to arrays from pointers to non-array objects, and preserving the pointer information within the compiler can ease the job of enforcing memory safety.

We conclude that it is possible to enforce comprehensive memory safety with low performance overheads using pointer-based checking with disjoint metadata and an efficient streamlined implementation. From our experience in building pointer-based checking in various parts of the tool chain, we anticipate that hardware acceleration (like Intel MPX) will reduce the performance overheads significantly while reducing the hardware changes. Intel MPX with support for incremental deployment of memory safety checking and its store-only instrumentation will likely make always-on deployment of spatial safety enforcement a reality.

Acknowledgments. We thank Vikram Adve, Emery Berger, John Criswell, Chrisitan Delozier, Joe Devietti, Michael Hicks, David Keaton, Peter-Michael Osera, Stelios Sidiroglou-Douskos, Nikhil Swamy, and Adarsh Yoga for their inputs during the course of this project. This research was funded in part by Intel Corporation, and the U.S. Government by DARPA contract HR0011-10-9-0008, ONR award N000141110596 and NSF grants CNS-1116682, CCF-1065166, CCF-0810947, and CNS-1441724. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

1 Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

2 Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

3 Stephen Bradshaw. Heap spray exploit tutorial: Internet explorer use after free aurora vulnerability. `http://www.thegreycorner.com/2010/01/heap-spray-exploit-tutorial-internet.html`.
4 David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.
5 Weihaw Chuang, Satish Narayanasamy, and Brad Calder. Accelerating meta data checks for software correctness and security. *Journal of Instruction Level Parallelism*, 9, June 2007.
6 Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, 2007.
7 Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.
8 Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the Foundations of Intrusion Tolerant Systems*, pages 227–237, 2003.
9 John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.
10 Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M.K. Martin, and Steve Zdancewic. IroncladC++: A Library-augmented Type-safe Subset of C++. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'13, 2013.
11 Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
12 Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 162–171, 2006.
13 Yulei Sui Ding Ye, Yu Su and Jingling Xue. Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *Proceedings of the 25th IEEE Symposium on Software Reliability Engineering*, 2014.
14 Frank Ch. Eigler. Mudflap: Pointer Use Checking for C/C++. In *GCC Developer's Summit*, 2003.
15 Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, 2015.
16 Kittur Ganesh. *Pointer Checker: Easily Catch Out-of-Bounds Memory Accesses.* Intel Corporation, 2012. `http://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf`.
17 Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. Architectural support for low overhead detection of memory viloations. In *Proceedings of the Design, Automation and Test in Europe*, 2009.

18 Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter Usenix Conference*, 1992.

19 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

20 Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-022 edition, October 2014. `https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf`.

21 Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

22 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*, 1997.

23 Greg Kroah-Hartman. The linux kernel driver model: The benefits of working together. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, Inc., June 2007.

24 Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

25 James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

26 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, page 75, 2004.

27 Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Internet Society Symposium on Network and Distributed Systems Security*, 2015.

28 Nuno Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

29 Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

30 Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995.

31 Santosh Nagarakatte. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. PhD thesis, University of Pennsylvania, 2012.

32 Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

33 Santosh Nagarakatte, Milo M K Martin, and Steve Zdancewic. Hardware-enforced comprehensive memory safety. In *IEEE MICRO 33(3)*, May/June 2013.

34 Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO'14*, page 175, 2014.

35 Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

36 Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, 2010.

37 George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

38 Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2004.

39 Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 65–74, 2007.

40 Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

41 Harish Patil and Charles N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. In *Software Practice and Experience 27(1)*, pages 87–110, 1997.

42 J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. In *IEEE Security and Privacy 2(4)*, pages 20–27, 2004.

43 Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, February 2009.

44 Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, pages 291–302, 2005.

45 Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 159–169, February 2004.

46 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

47 Matthew S. Simpson and Rajeev Barua. Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. In *Software Practice and Experience, 43(1)*, 2013.

48 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

49 Guru Venkataramani, Brandyn Roemer, Milos Prvulovic, and Yan Solihin. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, pages 273–284, 2007.

50 David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2000.

51 Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating System Principles*, 2013.

52 Wei Xu, Daniel 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*, pages 117–126, 2004.

53 Suan Hsi Yong and Susan 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)*, pages 307–316, 2003.