WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking

Santosh Nagarakatte
Rutgers University
santosh.nagarakatte@cs.rutgers.edu

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

Abstract

Lack of memory safety in C is the root cause of a multitude of serious bugs and security vulnerabilities. Numerous software-only and hardware-based schemes have been proposed to enforce memory safety. Among these approaches, pointer-based checking, which maintains per-pointer metadata in a disjoint metadata space, has been recognized as providing comprehensive memory safety. Software approaches for pointer-based checking have high performance overheads. In contrast, hardware approaches introduce a myriad of hardware structures and widgets to mitigate those performance overheads.

This paper proposes WatchdogLite, an ISA extension that provides hardware acceleration for a compiler implementation of pointer-based checking. This division of labor between the compiler and the hardware allows for hardware acceleration while using only preexisting architectural registers. By leveraging the compiler to identify pointers, perform check elimination, and insert the new instructions, this approach attains performance similar to prior hardware-intensive approaches without adding any hardware structures for tracking metadata.

Categories and Subject Descriptors C.1 [*Computer Systems Organization*]: Processor Architectures; D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance, Security

Keywords memory safety, spatial safety, temporal safety, bounds checking, use-after-free checking

1. Introduction

C and C++ are the languages of choice for implementing infrastructure code and all kinds of low-level software. Such languages remain in common usage both for legacy reasons and because they provide low-level access to underlying hardware, explicit control over memory management, and high performance. However, a longstanding problem with code written in C/C++ is the lack of memory safety: accessing beyond the bounds (spatial safety violations) and accessing unallocated/deallocated memory locations (temporal safety violations). The lack of memory safety causes simple programming errors to become the root cause of a multitude of memory corruption bugs and security vulnerabilities [7, 37, 38].

CGO '14 February 15-19 2014, Orlando, FL, USA

http://dx.doi.org/10.1145/2544137.2544147

Efficiently and comprehensively detecting and protecting against memory safety violations is unsurprisingly a well researched topic with numerous proposals over the years [1, 2, 4, 9–14, 27, 29, 31, 36]. These include both software-only tools [1, 2, 4, 8, 11, 13, 14, 27, 29, 31] and hardware instantiations [9, 10, 12, 25, 36]. Beyond academia, recent tools from industry—such as Google's Address Sanitizer [40] in the LLVM compiler and Intel's Pointer Checker compiler [15], patent application [35], and recently announced MPX ISA extensions [19]—illustrate the importance of detecting memory safety violations.

Prior proposals for detecting memory safety violations provide a wide spectrum of protection ranging from partial countermeasures to comprehensive memory safety. These proposals make tradeoffs along the dimensions of performance, protection, and compatibility with existing applications. Szekeres *et al.* [43] surveyed the entire space of memory safety vulnerabilities and enforcement mechanisms and identified pointer-based checking as the only approach to provide comprehensive and non-probabilistic detection of memory safety vulnerabilities.

Pointer-based checking [2, 10, 12, 15, 27–29, 29, 36, 47] gives every pointer a view of memory that it can legally access by maintaining *per-pointer metadata*. To retain memory layout compatibility, some proposals place this per-pointer metadata in a disjoint-metadata space [12, 15, 27, 28]. The per-pointer metadata is propagated on pointer operations. Conceptually, every pointer dereference is checked using its metadata.

Comprehensive memory safety requires detecting both spatial (bounds) violations and temporal (dangling pointer or use-after-free) violations. To detect spatial safety violations, base and bounds metadata is maintained with each pointer. Temporal violations may be detected using unique identifier based checking on memory accesses [2, 10, 25, 28, 36, 47] or by invalidating the bounds of all pointers to an object when deallocating the object [15, 16, 42] so that subsequent bounds checks will fail.

Pointer-based checking can be implemented in various parts of the system stack—via source code rewriting, the compiler, and/or in hardware. Recent compiler-based implementations have reduced the performance overhead for comprehensive memory safety to approximately 2× on average. These overheads are attained by instrumenting optimized code and using information available to the compiler. Unfortunately, this overhead is likely still too large for production use. As a consequence, researchers have proposed using hardware to accelerate pointer checking [10, 12, 16, 25, 35], but these hardware proposals—including Watchdog [25], our own prior proposal—introduce significant hardware complexity and require various hardware structures dedicated to recording metadata state. See Section 2 for a comparison of these strategies.

This paper proposes WatchdogLite, an ISA extension to accelerate pointer-based checking without adding any new hardware for maintaining metadata state. The proposed instructions accelerate the three key memory-safety checking operations: loading and storing metadata, bounds checking, and use-after-free checking. The instructions operate on the ISA's preexisting architectural registers.

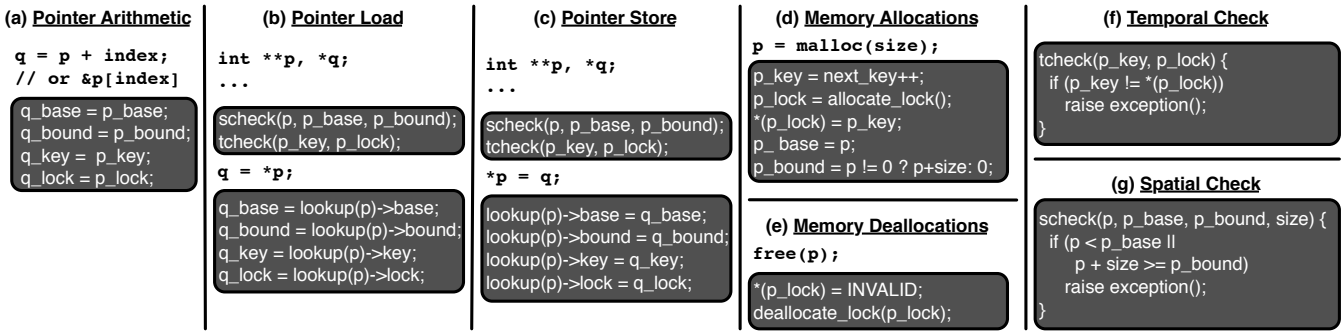

Figure 1. (a) Pointer metadata propagation with pointer arithmetic, (b) metadata propagation through memory with metadata lookups on loads, (c) metadata lookups with pointer stores, (d) pointer metadata creation on memory allocations, (e) identifier metadata being invalidated on memory deallocations, (f) lock and key checking using identifier metadata, and (g) spatial check performed using bounds metadata.

The compiler explicitly inserts these instructions, uses pre-existing static optimizations to eliminate many checks, and performs in-register metadata propagation by copy elimination and standard register allocation. Relying on the compiler to perform these tasks largely eliminates the need for various previously proposed dedicated hardware structures that track and cache metadata.

Experiments based on extensions to our SoftBound+CETS compiler instrumentation show that the performance overhead for enforcing comprehensive memory safety is reduced on average from 90% (without hardware acceleration) to 29% (with the new instructions). This overhead is similar to prior hardware schemes, which use extensive hardware structures to track and propagate metadata state, which indicates that the proposed ISA extension is a more pragmatic approach for hardware acceleration of memory safety enforcement than prior hardware-centric proposals [12, 25, 35].

Concurrent with this work, Intel developed Memory Protection Extensions (MPX) [19] and released the ISA specification in 2013. The work described in this paper was largely completed in 2012 (see Chapter 6 of [24]). The WatchdogLite ISA extensions described in this work and Intel's MPX are similar in many ways, including: using pointer-based checking with disjoint metadata, adding new instructions for efficiently accessing the metadata shadow space, and adding instruction for accelerating bounds checking. One difference is that MPX does not include support for accelerating use-after-free checking. The differences and similarities are discussed further in Section 5.

2. Background on Pointer Checking

This section provides background on the pointer-based checking approach [2, 10, 12, 15, 27–29, 29, 36, 47] and describes various prior proposals for implementing it in either software or hardware. Although there are many memory-checking proposals, this discussion focuses on pointer-based checking with disjoint metadata [12, 15, 27, 42] for several reasons. This approach has been shown to be highly compatible with existing code [12, 15, 27, 42], a recent paper surveying the entire space of memory vulnerabilities [43] identified this approach as the only one to provide comprehensive and non-probabilistic detection of memory safety vulnerabilities (see Table II of Szekeres *et al.* [43]), it has been formally shown to provide strong memory safety properties [48], and it has been embraced by Intel in a recently released commercial software product [15] and a patent application [35].

2.1 Pointer-Based Checking with Disjoint Metadata

Enforcing *memory safety* prevents memory corruption bugs and prevents the entire class of memory corruption vulnerabilities [43]. Such vulnerabilities—including buffer overflows and use-after-free vulnerabilities—are still pervasive but they are not new [37, 43]. Informally, enforcing memory safety has two primary components: preventing spatial violations (out-of-bounds memory accesses and buffer overflows of all sorts) and preventing temporal safety violations (memory accesses to deallocated memory, a.k.a. dangling pointer or use-after-free violations).

Pointer-based metadata. In a pointer-based approach, metadata is maintained with each pointer, providing it a view of the memory that it can safely access according to the language specification. 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++). Figure 1 illustrates the pointer-based metadata propagation and checking abstractly using pseudo C code notation. 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 1a).

The per-pointer metadata may be maintained inline, as in fat-pointer approaches [2, 10, 29], or in a disjoint metadata space for pointers in memory [12, 16, 25, 27, 28]. Using a disjoint metadata space protects the metadata from malicious corruption and leaves the memory layout of the program intact, retaining compatibility with existing code. With disjoint metadata, the metadata is loaded/stored from the disjoint metadata shadow space whenever a pointer is loaded/stored (Figure 1b and Figure 1c, respectively). This disjoint metadata space can be implemented in various ways, including a linear region of memory [12, 25, 27], a hash table [27], or a trie data structure [15, 28, 30].

Spatial checking. To enforce spatial safety, the base and bound of the legal region of the memory accessible via the pointer is associated with the pointer when it is created. The base and bound are each typically 64-bit values, so they can encode arbitrary byte-granularity bound information. These per-pointer base and bounds metadata fields are sufficient to perform a bounds check prior to a memory access (Figure 1g).

Temporal checking. To enforce temporal safety, a unique identifier is associated with each memory allocation (Figure 1d). 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 pointers.[1] On a pointer dereference, the system checks that the unique allocation identifier associated with the pointer is still valid.

[1] Szekeres *et al.* [43] note: "The only way to detect a use-after-free attack reliably is to associate the temporal information with the pointer and not with the object."

	Safety checking	Instrumentation method	Metadata organization	Avoids new arch. state?	Static check optimization?	Checking method	Performance overhead
Chuang *et al.* [10]	Spatial & Temporal	Compiler + Hardware	inline (fat pointers)	No	No	Implicit	30%
HardBound [12]	Spatial	Hardware	disjoint (shadow space)	No	No	Implicit	5-9%†
SafeProc [16]	Spatial & Temporal	Compiler	disjoint (256-entry CAM)	No	Yes*	Explicit	93%‡
Watchdog [25]	Spatial & Temporal	Hardware	disjoint (shadow space)	No	No	Implicit	25%
Intel's MPX [19] (concurrent work)	Spatial	Compiler	disjoint (two-level trie)	No	Yes*	Explicit	N/A
WatchdogLite (this work)	Spatial & Temporal	Compiler	disjoint (shadow space)	Yes	Yes	Explicit	29%

†HardBound uses a special low-overhead encoding for small objects and does not perform temporal checking.

*These proposals benefit from static check optimization, but results with such optimizations are not reported.

‡The SafeProc paper reports performance overheads as low as 9%, but only when checks are delayed and queued into a 256-entry FIFO memory updated buffer (MUB) and executed by a separate dedicated controller.

Table 1. Comparison of hardware implementations of pointer-based checking schemes along the following dimensions: type of safety checking (spatial vs temporal), instrumentation method, metadata organization, avoiding new architectural state, employing static check optimizations in the compiler, checking method (implicit vs explicit), and performance overhead.

Performing a validity check on each memory access using a hashtable or splay tree can be expensive [2, 20], 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* [10, 25, 28, 36, 47]. 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 1f). Freeing an allocated region changes the value at the lock location, thereby invalidating any other (now-dangling) pointers to the region (Figure 1e). Because the keys are unique, a lock location itself can be reused after the space it guards is deallocated.

2.2 Compiler Implementations of Pointer Checking

The pointer-based checking described above can be implemented on either the source code of the program with source-to-source translation [29, 47] or within the compiler [15, 27, 28, 42]. Compiler-based implementations provide three primary 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. By implementing pointer-based checking within the compiler, such approaches [15, 27, 28, 42] have reduced the performance cost of enforcing comprehensive memory safety to approximately 2× performance overhead on average, but this is likely still too costly for widespread use in production code.

2.3 Hardware Implementations of Pointer Checking

To overcome the performance overheads of software-only approaches, researchers have proposed implementing pointer-based checking primarily in hardware [10, 12, 16, 25, 35]. There are three main dimensions that distinguish these proposals: (1) whether they perform implicit or explicit checking, (2) how they identify pointers, and (3) the organization of the in-memory metadata. These design decisions have implications on the safety provided, the hardware structures required, and how well they can be optimized. Table 1 compares the prior hardware proposals on these dimensions. Table 2 lists the hardware structures used by the proposals.

	Hardware Structures
Chuang *et al.* [10]	(1) μop injection (2) 32-entry metadata check table (3) Metadata base register map (for each register)
HardBound [12]	(1) μop injection (2) Pointer tag cache accessed on each memory access
SafeProc [16]	(1) 256-entry hardware CAM (associatively searched on every memory access check) (2) Hardware hash table (3) 256-entry FIFO update buffer
Watchdog [25]	(1) μop injection (2) lock location cache used on each memory access (3) changes to the register renamer

Table 2. Hardware structures used by various hardware approaches. A CAM is a content addressable memory.

Hardware with implicit checking uses μop injection to check and propagate metadata. Prior proposals such as the work of Chuang *et al.* [10], HardBound [12] and Watchdog [25] fall into this category. To mitigate the performance overhead of performing extra metadata accesses on every memory operation in such approaches, various pointer tag caches [12] and special caches for lock locations [25] have been proposed. Although implicit checking can be efficiently implemented by introducing such hardware structures, these proposals have not leveraged the benefits of static check optimizations. In contrast, explicit checking approaches such as SafeProc [16] modify the software tool-chain to insert instructions. Although the evaluation in the SafeProc paper did not leverage static check elimination in the compiler, its explicit checking approach would allow compilers to eliminate checks while preserving the hardware acceleration benefits for the remainder.

A hardware pointer-checker needs to identify pointers. This information is typically absent in binaries. HardBound [12] accesses

the metadata on every memory access, but introduces a hardware pointer tag cache to reduce the cost of such accesses in the common case. Watchdog [25] uses conservative heuristics to reduce metadata accesses by filtering out all non-pointer-sized memory operations. SafeProc [16] and the work of Chuang *et al.* [10] provide ISA extensions to allow the compiler to precisely identify pointer operations.

The metadata organization is an important aspect of providing comprehensive safety. The fat pointers in the work of Chuang *et al.*—like any proposal that uses inline metadata [29]—can be corrupted in the presence of arbitrary type casts. Approaches that use disjoint metadata either as a shadow space or a hardware table prevent such corruption, allowing for comprehensive protection. In Chuang *et al.* metadata may only reside in memory (not registers or other hardware structures), so each check operation is fairly expensive as it must load all the metadata from memory into the core as part of performing bounds and use-after-free checking (adding approximately four memory accesses per check, and checks are by default performed on every memory access).

Among the prior approaches, the explicit checking performed by SafeProc [16] is most closely related to the proposal described in this paper. SafeProc uses ISA extensions to identify pointers and insert checks. However, SafeProc uses *bounds invalidation* to detect temporal safety violations; when an object is deallocated, the system must find all of the pointers that point to the object and set their bounds information to invalid so that subsequent accesses will fail [16, 42]. To implement this approach, SafeProc places all per-pointer metadata into a 256-entry hardware CAM (content addressable memory) that is associatively searched on every memory access check (matching on pointer address) and on every object deallocation (matching on object address). As programs can have more than 256 pointers, the SafeProc hardware relocates pointer records that overflow the CAM into an in-memory dual-indexed hash table data structure that supports hardware walking of the data structure to lookup the record for a pointer address (used to perform bounds checking) or all the pointer records that are associated with a particular object address (used to perform object deallocation), which in turn necessitates other hardware extensions like the memory update buffer [16] to mitigate performance overheads.

The goal of WatchdogLite is to provide memory safety acceleration using preexisting structures without adding any hardware structures that maintain state. To accomplish this goal, WatchdogLite (1) uses explicit checking, which enables the compiler to perform custom check optimizations eliminating the need for some of the hardware structures (*e.g.* lock location cache, μop injection), (2) employs hardware-accelerated lock-and-key checking for detecting use-after-free violations rather than using object bounds invalidation, and (3) relies on the compiler to perform metadata propagation, copy elimination, and check elimination. This hardware/software co-design approach has the potential to provide the low performance overheads of prior hardware-intensive proposals without introducing any new hardware structures for recording metadata state.

3. Instructions that Accelerate Checking

This paper proposes WatchdogLite, a set of instructions to provide hardware acceleration for three key memory safety checking operations: (1) loading and storing metadata, (2) bounds checking, and (3) use-after-free checking. Unlike the few prior proposals for hardware acceleration of pointer checking [10, 12, 25, 35], the proposed instructions operate only on values in pre-existing ISA architectural registers—they do not add any extra lookup tables, buffers, caches, sidecar/shadow registers, *etc.*, nor do they require mechanisms needed by prior schemes, such as μop injection [10, 25, 35], copy elimination register renaming tricks [25], or replicated shadow datapaths with metadata accesses on every memory operation [12].

The key to this approach is to leverage the compiler to avoid the aforementioned extra hardware state and mechanisms. WatchdogLite does this by establishing a division of labor between the compiler and the hardware that harnesses the complementary benefits of prior compiler-based and hardware-based approaches. The compiler generally knows which operations are manipulating pointers, so it can insert operations to perform metadata manipulation only for those operations that actually require it. Instead of in-register pointer manipulations resulting in explicit copying of metadata [10, 12, 16] or relying on dynamic copy elimination [25], the compiler can simply employ its existing static copy propagation optimization. The compiler is already responsible for managing scarce register resources via standard register allocation. In addition, it has long been known that compilers can reduce checking overhead by removing unnecessary or redundant checks [23].

Our hypothesis is that compiler-based implementations will benefit from new instructions that perform these common checking and metadata shadow space operations more efficiently than building them from individual instructions, providing performance similar to prior hardware-intensive approaches with much less invasive hardware modifications. Overall, the instructions are intended to be straightforward to understand and similar in implementation complexity to adding a few additional SSE2/AVX or VFPv4/NEON instructions, which have been added over the years to x86 and ARM, respectively.

We investigate two variants of the proposed instructions: (1) instructions that operate on 64-bit "narrow" registers and (2) variants that operate on 256-bit "wide" registers. In the first variant, the instructions use only the preexisting 64-bit general-purpose registers and the compiler uses its standard register allocation to assign general-purpose registers for holding each of the four words of metadata per pointer. This variant is similar to how conventional software-only checking schemes [15, 27] operate, and so it provides a natural point of experimental comparison.

The second variant further reduces overhead by leveraging preexisting 256-bit AVX "wide" registers found in today's Core i7 x86 processors (`%YMM0-%YMM15`), packing the four words of metadata into a single wide register. Although originally proposed for floating point and vector operations, the x86 SSE2/AVX family of extensions also accelerates XML parsing, cyclic redundancy checks, and encryption. Thus we explore a "wide" variant of the memory-checking acceleration instructions that further exploits these registers. Packing the metadata into a single wide register has two main advantages: it reduces register pressure on the general-purpose registers and it allows all four words of metadata to be loaded/stored in a single 256-bit wide aligned cache access (rather than one cache access per word of metadata).

3.1 Metadata Load and Store Instructions

The *MetaLoad* and *MetaStore* instructions accelerate loading and storing the four 64-bit words of per-pointer metadata from the disjoint shadow space when loading and storing pointers from/to memory. Each time a pointer is loaded from memory into a register, the compiler also inserts a *MetaLoad* instruction to bring the pointer's metadata into registers. Similarly, the compiler inserts a *MetaStore* instruction following each store of a pointer to memory.

The location of the metadata in the shadow space is determined by the address from which the pointer is being loaded. Similar to a normal memory operation, the *MetaLoad* and *MetaStore* instructions support a "register plus offset" addressing mode to specify the address. The *MetaLoad* instruction also specifies the destination register for the loaded metadata; the *MetaStore* instruction specifies the source register that holds the metadata to be written to the

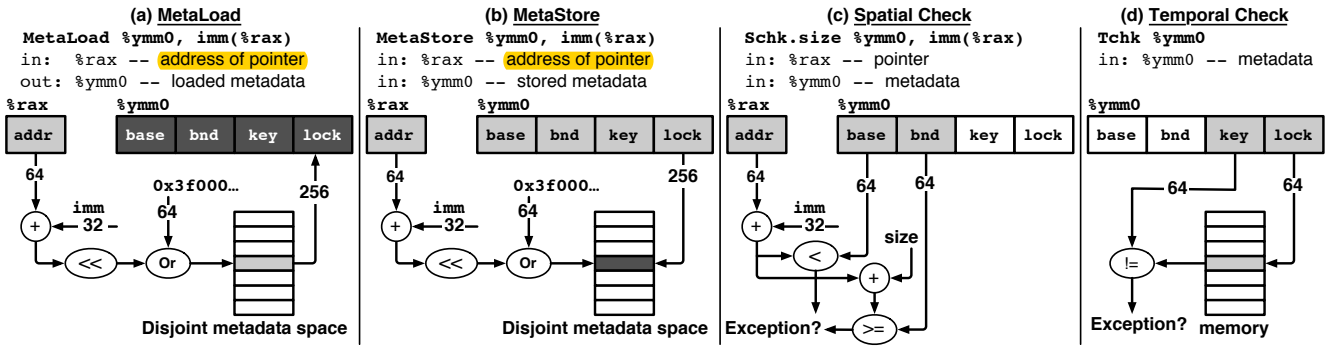

Figure 2. Operation of the *MetaLoad* (a), *MetaStore* (b), *SChk* (c), and *TChk* (d) instructions.

shadow space. These new instructions combine the shadow space address generation/mapping operations with the memory access to the shadow space into a single instruction.

The *MetaLoad* and *MetaStore* instructions have two variants. When the metadata is held in four individual 64-bit registers, each *MetaLoad* instruction loads one 64-bit word of metadata from the shadow space into the register file (sub-opcode bits in the instruction indicate which of the four words to access); when the per-pointer metadata is packed into a single 256-bit register, a *MetaLoad* instruction uses a single 256-bit wide cache access.

Shadow space metadata mapping. As in prior hardware shadow space implementations [12, 25], the implementation of these instructions assumes that the shadow space is a linear address range mapped into a fixed location in the upper regions of the virtual address space. (Alternatively, the start of the shadow memory could be specified by the value in a control register, much like the base of the page table.) This linear shadow space is more efficient than using hash tables [27] or trie data structures [15, 28] to implement the shadow space, but it requires hardware and/or operating system support.

Without hardware support, the operations to perform the mapping can add significant performance overhead. The instruction sequence generated by the compiler to perform metadata loads/stores using a two-level trie data structure incurs about a dozen x86 instructions, and even a linear shadow space encoding requires a few shift/mask/and instructions to map the pointer address to the address of the corresponding shadow mapping. The *MetaLoad* and *MetaStore* instructions hardcode these manipulations and perform these bit manipulations internally using custom hardware as part of the address generation stage. Figure 2(a) and Figure 2(b) summarize the instruction interface and implementation.

3.2 Spatial (Bounds) Check Instruction

To accelerate bounds checks, we add a new instruction, *SChk*, the *spatial check* instruction. This single instruction replaces the five x86 instructions (cmp, br, lea, cmp, br) that perform a (lower and upper) bounds check. The *SChk* instruction requires three values: the address being checked, the base, and the bound. Opcode variants of this instruction encode the width of the access being checked (in powers of two ranging from one to 32 bytes). When the check fails, the instruction raises an exception.

The *SChk* instruction reads three 64-bit values from the register file. It performs two parallel comparisons: the base with the pointer and the bound with the pointer plus the access size. Because the size of the access may be only 1/2/4/8/16/32 bytes, a fully general adder is not required. This instruction does not produce output, so the latency of the instruction need not be a single-cycle to obtain low performance overheads.

The *SChk* instruction has two variants. The narrow variant uses three 64-bit general purpose registers as inputs (address, base, and bound). The wide variant reads the pointer value from a 64-bit register and the base and bound from consecutive elements of a single 256-bit wide register. Figure 2(c) summarizes the instruction interface and provides the details of the implementation.

Comparison to the x86 bounds instruction. *SChk* is similar in spirit to the bound instruction available on x86 processors since the 80286, but *SChk* is different in two key ways. First, *SChk* uses registers to hold all inputs. In contrast, early x86 processors had just eight registers and no double-word datapaths, so the *bound* instruction required both the base and bound to be fetched from memory for each check. As a result, when the base and bound were already in registers, the *bound* instruction was typically more expensive than a bounds check using other x86 instructions. Second, *SChk* efficiently supports the byte-granularity checking used by bounds checks. Byte-granularity checking provides the ability to prevent a four-byte memory access to a three-byte object, but not flag a two-byte access to the same address. *SChk* facilitates this by encoding the size of the memory access. In contrast, the x86 *bound* instruction was designed for checking at the granularity of an array index. Using *bound* would require additional instructions to adjust the upper bound based on the size of the memory access.

3.3 Temporal (Dangling Pointer) Check Instruction

The *TChk* instruction accelerates the "lock and key" temporal checks described in Section 2. A *TChk* instruction replaces three x86 instructions (load, compare, branch). The instruction reads the 64-bit key and 64-bit lock value from the register file, performs a 64-bit load using the lock part of the input, and checks that the loaded value is equal to the key. When the check fails, it raises an exception.

If the memory access datapath is extended to perform a post-load comparison, this instruction can execute as a single μop. An alternative implementation is to crack it into two μops: a load μop and a new compare-and-fault μop. The *TChk* instruction does not produce an output register, so performance is not particularly sensitive to the instruction's execution latency.

We evaluate two variants of the *TChk* instruction. The narrow variant uses two 64-bit register inputs for the lock and key. The wide variant obtains the two 64-bit words from elements of a single 256-bit wide register. Figure 2(d) summarizes the instruction interface and provides the details of the implementation.

4. Experimental Evaluation

This section provides an experimental evaluation of pointer checking using the proposed instructions, highlights its effectiveness in attaining low performance overheads, and analyzes the sources of the performance overhead. The next subsection describes our experimental setup, the benchmarks, changes made to the entire tool chain—compiler, assembler, and the simulator—and the experimental methodology.

4.1 Experimental Methodology

Baseline compiler prototype for software-only checking. We use the LLVM-based SoftBound+CETS prototype for our experiments, both as a point of experimental comparison and to insert the proposed instructions. The compiler performs the standard suite of optimizations on the LLVM intermediate representation (IR) prior to inserting metadata propagation and checking operations. The prototype instruments code by adding calls to helper functions written in C that are later forcibly inlined prior to rerunning the full suite of optimization passes.

To avoid changing the calling convention for functions, the prototype implements a shadow stack that mirrors the call stack of the program to pass/return metadata for pointer arguments to function calls. The shadow stack also provides a mechanism for dynamic typing between the arguments pushed at the call site and the arguments retrieved by the callee preventing memory safety errors with variable argument functions and calls with mismatched function signatures [24].

The compiler elides many unnecessary checks (*e.g.*, bounds checking of scalar local variables or stack spill/restores) and performs a simple intra-procedural dominator-based redundant check elimination. The prototype compiler does not implement more sophisticated loop-based or constraint-based check eliminations [6], which would further reduce its performance overhead.

Modifications to prototype compiler and assembler. We modified the assembler and other binutils to accept the new instructions. For the "narrow" variant of the new instructions, we simply replaced the aforementioned helper functions written in C with inline assembly to invoke the new instructions. Supporting the "wide" variant required changing the compiler to associate a single wide temporary in the LLVM IR with each pointer rather than four narrow registers. In both variants, using inline assembly to insert the new instructions was not sufficient to utilize the "register plus offset" addressing mode, resulting in many `LEA` instructions being generated prior to check operations. For example, bounds checking of the following memory access:

```
movq %rax, 8(%rbx)
```

translates into:

```
movq %rax, 8(%rbx)
lea %rcx, 8(%rbx)
SChk.q 0(%rcx), %ymm1
```

whereas, it would ideally just be:

```
movq %rax, 8(%rbx)
SChk.q 8(%rbx), %ymm1
```

We observe an increase in `LEA` instructions roughly proportional to the number of bound check instructions (see Section 4.4), which is consistent with most spatial check instructions being preceded with an instruction to calculate the effective address.

Simulator. To evaluate the benefits of the WatchdogLite ISA extensions, we use an in-house x86-64 simulator developed for microarchitecture research [17, 18]. The simulator executes the user-level portions of statically linked 64-bit x86 programs. It decodes x86 macro instructions and cracks them into a RISC-style μop ISA. Table 3 describes the configurations used for each component of the micro-architecture. The out-of-order processor configurations described are designed to be similar to Intel's Core i7 "Sandy Bridge" processor. We model the details of Core i7 using the publicly available information such as the memory hierarchy (large L3 cache split into banks on a ring interconnect with private L2/L1), and structure sizes (ROB, LQ, SQ, IQ, *etc.*). We simulate a three level cache hierarchy with private L1 and L2 caches of size 32KB and 256KB respectively and a shared L3 cache of size 16MB divided into four banks organized as a ring.

Front-end	Clock	3.2 GHz
	Bpred	3-table PPM: 256x2, 128x4, 128x4, 8-bit tags,2-bit counters
	Fetch	16 bytes/cycle. 3 cycle latency
	Rename	Max 6μops per cycle. 2 cycle latency
	Dispatch	Max 6μops per cycle. 1 cycle latency
Window/Exec	Registers	(160 int + 144 floating point), 2 cycle
	ROB/IQ	168-entry ROB, 54-entry IQ
	Issue	6-wide. Speculative wakeup.
	Int FUs	6 ALU. 1 branch. 2 ld. 1 st. 2 mul/div
	FP FUs	2 ALU/convert. 1 mul. 1 mul/div/sqrt.
	LSQ size	64-entry LQ, 36-entry SQ
Memory Hierarchy	L1 I$ Prefetcher	32KB. 4-way, 64B blocks. 3 cycles 2-streams, 4 blocks each
	L1 D$ Prefetcher	32KB, 8-way, 64B blocks, 3 cycles 4-streams, 4 blocks each
	L1 $\leftrightarrow$ L2 bus	32-bytes/cycle. 1 cycle.
	Private L2$ Prefetcher	256KB, 8-way, 64B blocks, 10 cycles. 8 streams. 16 blocks.
	L2 $\leftrightarrow$ L3 bus	8-stop bi-directional ring. 8-bytes/cycle/hop. 2.0GHz clock
	Shared L3$	16MB. 16-way, 64B blocks, 25 cycles
	Mem. Bus	800MHz. DDR. 8-bytes wide. Dual channel. 16ns latency

Table 3. Simulated processor configurations.

Benchmarks. We used the fifteen C SPEC benchmarks from the SPEC2006 and SPEC2000 benchmark suites. We compiled the benchmarks using the LLVM compiler version 3.0 with aggressive analysis-based optimizations including SSE-4.2 related optimizations. Hence, the floating point benchmarks use the `XMM/YMM` registers for floating point values (and do not use the floating point stack).

To ensure reasonable simulation times, we use train/test inputs with 2% periodic sampling with each sample of 10 million instructions proceeded by a fast forward and a warmup of 480 and 10 million instructions per period, respectively. The number of samples executed varies across benchmarks (from a minimum of 32 samples to a maximum of 384 samples per benchmark). Based on the SMARTS methodology [46], we calculated that this sampling introduces 95% confidence intervals of approximately 1% to reported execution times. Execution times are calculated using the macro instruction IPC (instructions per cycle) and the number of instructions executed. The instruction counts reported are not sampled and thus have no sampling error.

4.2 Functional Evaluation

To evaluate the effectiveness in preventing bounds errors, we ran multiple test suites: NIST Juliet Test Suite for C/C++ [32], SAFECode test suite, and Wilander test suite [45]. These include more than 2000 test cases exercising various kinds of buffer overflows. To evaluate the effectiveness with use-after-free vulnerabilities, we ran 291 test cases for use-after-free vulnerabilities (CWE-416 and CWE-562) from the NIST Juliet Test Suite for C/C++ [32], which are modeled after various use-after-free errors reported in the wild. It successfully detected and prevented both buffer overflows and use-after-free vulnerabilities without any false positives.

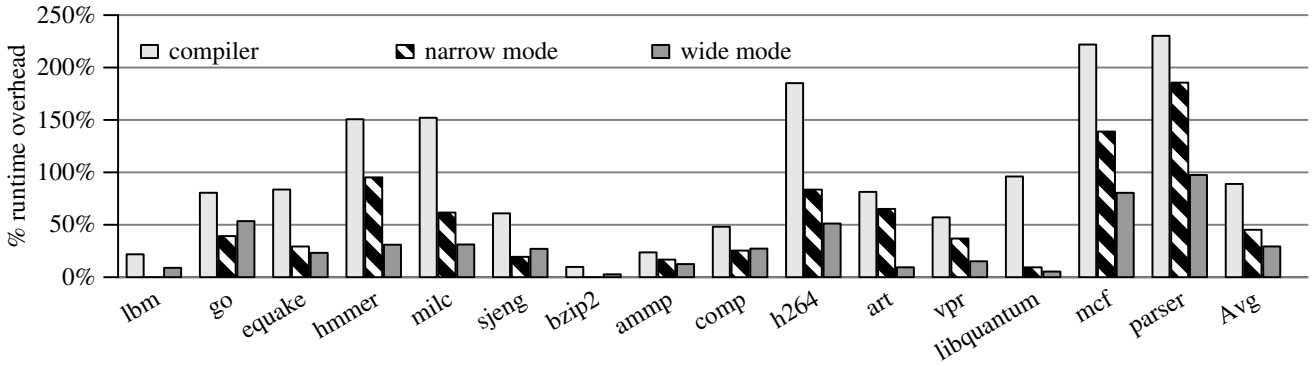

Figure 3. Performance overhead with compiler, ISA extension in narrow (scalar) mode, and ISA extensions in wide mode.

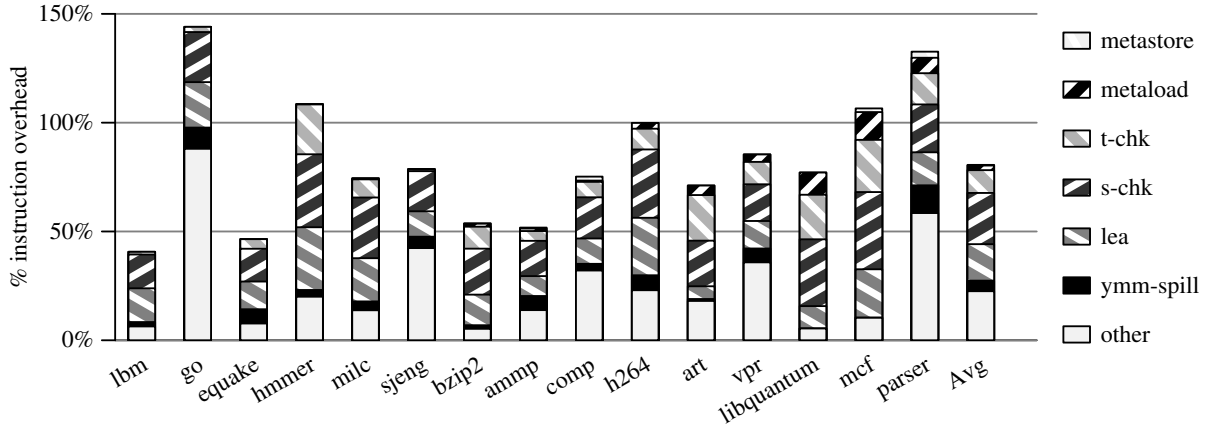

Figure 4. Instruction overhead breakdown in wide mode.

4.3 Runtime Execution Overheads

Figure 3 presents the percentage execution time overhead of pointer-based checking with and without the proposed instructions over a baseline without any memory safety instrumentation (smaller bars are better as they represent lower runtime overheads). The benchmarks are sorted by frequency of pointer metadata loads and stores (less frequent on the left to more frequent on the right). The graph contains three bars for each benchmark. The height of the leftmost bar represents the overhead of compiler-based instrumentation. The compiler instrumentation incurs an overhead of 90% on average for these benchmarks to provide comprehensive memory safety, which is similar to the overheads reported by prior work on compiler-based pointer checkers [24, 27, 28].

The height of the middle and rightmost bars present the performance overhead with both the narrow and wide variants of the proposed instructions (45% and 29% on average, respectively). The wide variant of the instructions improves performance not only by accelerating the checks but also by reducing the integer register pressure and the number of loads/stores performed with metadata loads/stores. The benchmarks on the right, which have a large number of metadata loads and stores, receive more benefit from the wide instructions.

4.4 Contributions to Performance Overhead

To understand the contribution of the various operations to the overall overhead, Figure 4 reports the breakdown of instruction overhead by type of instruction. The total height of the bar represents the total percentage increase in instructions over the unsafe baseline for the new instructions in wide register mode (81% more instructions on average, which results in a 29% average performance overhead). The percent increase in instruction overhead correlates well with the performance overheads in Figure 3, but note that the percent increase in instructions is larger than the percent increase in performance overhead; adding these off-the-critical-path instructions increases the ILP available in the program and adding such checking typically does not increase the time spent waiting on cache misses, thus lowering the memory stalls per instruction.

Metadata load and store instructions. The top two segments in each bar represent the contribution of *MetaStore* (1% on average) and *MetaLoad* (2% on average) instruction overhead, respectively. As each of these operations in the software-only baseline previously required approximately a dozen instructions, the availability of these new instructions successfully reduced the instruction overhead from a significant contributor (estimated to be approximately 35%) to the small single-digits.

Check instructions. The next two segments (third and fourth from the top) in each bar represent the contribution of *TChk* (11% on average) and *SChk* (23% on average) instruction overhead, respectively. There are fewer temporal checks than spatial checks, as static optimizations are more effective at removing them. *SChk* is the largest single contributor to the instruction overhead, and this overhead is understated because most of these *SChk* instructions are paired with an `LEA` or other instruction to calculate the effective

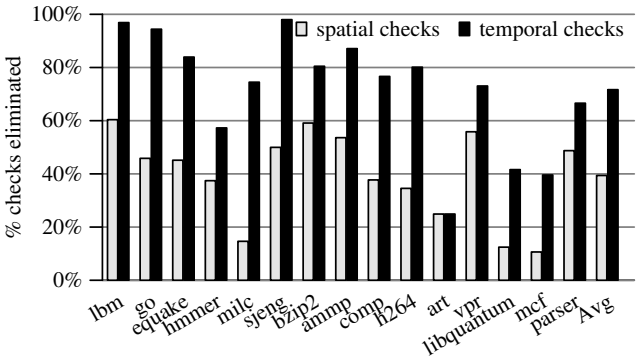

Figure 5. Percentage of memory access checks eliminated by static compiler optimizations.

address. We measured the increase in the number of `LEA` instructions versus the baseline (17% on average) and plotted it as the next segment (fifth from the top). Together, the *SChk* and extra address generation instructions represent about half of the instruction overhead (40% of 81%). The two most promising ways to further reduce this overhead are: (1) to modify the compiler's code generator to use the "register plus offset" addressing mode and (2) implement better bounds check elimination optimizations.

Additional spills/restores. Placing pointer metadata in the `%XMM/%YMM` registers increases register pressure, which could lead to more spill and restore operations on these registers. The segment second from the bottom represents the additional instructions that load or store `%XMM/%YMM` registers (5% on average). These additional operations are not negligible, but the additional register pressure is not the dominant source of overhead for any of these benchmarks.

Other overheads. The bottom segment of each bar is the remaining instruction overhead not accounted for by any of the prior categories (22% on average). Although we do not have an exact breakdown, this category includes additional instructions to establish a new lock and key on each function call, restore the lock and key on a return, and maintain the shadow stack for passing and returning per-pointer metadata on function calls [24]. The benchmarks `go`, `sjeng`, and `parser` all have significant "other" segments and also have high rates of function calls; in contrast, benchmarks such as `lbm` and `equake` perform few function calls and report fewer instructions in the "other" category. Changing the calling convention and stack layout would allow these operations to be performed more efficiently (by putting them on the main stack rather than a separate shadow stack), albeit at the risk of reducing compatibility with existing code and libraries.

Memory overheads. The memory overhead (due to shadow memory) is on average 56% for the benchmarks (unique physical pages touched, which are allocated on demand). These memory overheads are similar to prior disjoint pointer-based metadata schemes such as Watchdog [25] because the memory layout/mapping is identical.

4.5 Estimating the Benefit of Static Check Elimination

Unlike prior hardware proposals that perform implicit checking on every memory access [10, 12, 25, 25, 35], the instruction-based hardware acceleration described in this paper leverages compile-time static check elimination. To estimate the benefits of static check elimination, Figure 5 reports an estimate of the number of regular memory operations in each of the benchmarks that is not paired with a spatial check (left bar in each group) or temporal check (right bar in each group). On average, the static optimizations implemented in our compiler prototype eliminates 72% of the temporal checks and 40% of the spatial checks. Conservatively extrapolating the instruction counts from Figure 4, not using static check elimination would increase the number of temporal checks by at least 3.5× and the spatial checks by 1.6×. Together this would almost double (1.8×) the overall instruction overhead (from 81% to 147%).

The compiler's static check elimination assists WatchdogLite in obtaining performance (29% performance overhead) similar to our recently proposed hardware-injection scheme Watchdog [25, 26] (which reports 24% on average for spatial and temporal checking) without requiring the extra state and hardware structures introduced by Watchdog to accelerate metadata checking and propagation (*e.g.*, the lock location cache, μop injection, move elimination via modified register renaming, and a full suite of wide shadow registers). Our prototype compiler performs only simple check optimizations. A more sophisticated implementation would likely eliminate more checks and thus further reduce the overheads, potentially allowing WatchdogLite to outperform Watchdog while relying on simpler hardware.

5. Related Work

Memory safety enforcement for C/C++ is a well-researched topic. Apart from the techniques described in Section 2, there are other related approaches that seek to prevent memory safety errors either directly or indirectly. Please see the recent survey paper by Szekeres *et al.* [43] for a complete coverage of the efforts in this direction.

Hardware support for memory safety. Some hardware proposals, such as Memtracker [44], provide detection of some memory safety violations by maintaining valid/invalid metadata bits for each memory location and checking these bits before memory accesses [40, 41]. SafeMem [39] uses ECC to implement the valid/invalid metadata to detect some memory safety violations with low overhead. Log-based architectures (LBA) [9] accelerate valid/invalid status bit checking by executing the checks on a different core. LBA also provides idempotent filters and mapping instructions to reduce performance overheads.

Probabilistic mitigation. Probabilistic approaches randomize the locations of objects to make memory safety vulnerabilities more difficult to exploit in practice. Such approaches approximate an infinite heap with various kinds of randomization, including: instruction-set randomization [3, 21], address space randomization, and data randomization [5]. Other probabilistic approaches [4, 22, 33, 34] mitigate many memory safety vulnerabilities by allocating objects far apart and controlling memory reuse.

Intel MPX Extensions. Intel recently announced the specification of Memory Protection Extensions (MPX) [19] for providing hardware acceleration for compiler-based pointer-based checking with disjoint metadata. There are many similarities between Intel's concurrent work on MPX and WatchdogLite: both (1) provide hardware acceleration for compiler-based pointer-based checking, (2) use disjoint metadata for the pointers in memory, and (3) provide ISA support for efficient bounds checking. Differences between the proposals include: (1) MPX introduces four new multi-word bound registers (B0-B3) in contrast to reusing the existing wide AVX registers, (2) MPX extends interoperability 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, (3) MPX accesses the disjoint metadata using a two-level trie [28, 30], and (4) most significantly, MPX does not detect all memory safety errors, specifically use-after-free errors, whereas WatchdogLite provides comprehensive detection via lock-and-key use-after-free checking. Intel has thus far provided only the MPX ISA specification with no published evaluation or analysis of MPX.

6. Conclusion

This paper targets the elimination of an entire class of low-level bugs and security vulnerabilities by providing hardware acceleration of pointer-based comprehensive enforcement of memory safety. The proposed WatchdogLite ISA extension consists entirely of a few new instructions that operate on existing architectural registers. Even without the hardware search tables, sidecar registers, μop injection, and/or metadata caches used by prior hardware proposals, WatchdogLite attains low performance overheads. WatchdogLite accomplishes this with a division of labor between the compiler and the hardware: the compiler identifies pointers, propagates in-register metadata, eliminates unnecessary and redundant checks, and inserts the new instructions; the hardware simply provides a few new instructions to more efficiently perform the bounds checking, use-after-free checking, and storing/loading metadata to/from memory.

The most important contribution of this paper is therefore not necessarily the details of the proposed instructions, but rather our experimental finding that the various previously proposed hardware widgets and structures for metadata tracking and propagation are not fundamentally required to provide low-overhead enforcement of comprehensive memory safety.

Acknowledgments

We would like to thank Emery Berger for his comments and suggestions about this work and Andrew Hilton for use of his simulator. This research was funded in part by donations from Intel Corporation and the U.S. Government by ONR award N000141110596 and NSF grants CNS-1116682 and CCF-1065166. 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] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium*, Aug. 2009.

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

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

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

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

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

[7] S. Bradshaw. Heap Spray Exploit Tutorial: Internet Explorer Use After Free Aurora Vulnerability. `http://www.thegreycorner.com/2010/01/heap-spray-exploit-tutorial-internet.html`.

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

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

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

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

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

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

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

[15] K. 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`.

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

[17] A. Hilton and A. Roth. Decoupled Store Completion/Silent Deterministic Replay: Enabling Scalable Data Memory for CPR/CFP Processors. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010.

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

[19] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-015 edition, July 2013. `http://download-software.intel.com/sites/default/files/319433-015.pdf`.

[20] 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.

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

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

[23] V. Markstein, J. Cocke, and P. Markstein. Optimization of Range Checking. In *Proceedings of the 1982 SIGPLAN symposium on Compiler Construction*, 1982.

[24] S. Nagarakatte. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. PhD thesis, University of Pennsylvania, 2012.

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

[26] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Hardware-Enforced Comprehensive Memory Safety. *IEEE Micro*, 33(3), May/June 2013.

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

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

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

[30] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2007.

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

[32] NIST. *NIST Juliet Test Suite for C/C++*, 2010. `http://samate.nist.gov/SRD/`.

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

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

[35] B. V. Patel, R. Gopalakrishna, A. F. Glew, R. J. Kushlis, D. A. V. Dyke, J. F. Cihula, A. K. Mallick, J. B. Crossland, G. Nelger, S. D. Rodgers, M. G. Dixon, M. J. Charney, and J. Gottelieb. Managing and Implementing Metadata in Central Processing Unit Using Register Extensions, Mar. 2011. US Patent Pub No: US 2011/0078389 A1.

[36] 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.

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

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

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

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

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

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

[43] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

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

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

[46] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

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

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