Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety

Santosh Nagarakatte   Milo M. K. Martin   Steve Zdancewic

Computer and Information Science, University of Pennsylvania

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

Abstract

Languages such as C and C++ use unsafe manual memory management, allowing simple bugs (i.e., accesses to an object after deallocation) to become the root cause of exploitable security vulnerabilities. This paper proposes Watchdog, a hardware-based approach for ensuring safe and secure manual memory management. Inspired by prior software-only proposals, Watchdog generates a unique identifier for each memory allocation, associates these identifiers with pointers, and checks to ensure that the identifier is still valid on every memory access. This use of identifiers and checks enables Watchdog to detect errors even in the presence of reallocations. Watchdog stores these pointer identifiers in a disjoint shadow space to provide comprehensive protection and ensure compatibility with existing code. To streamline the implementation and reduce runtime overhead: Watchdog (1) uses micro-ops to access metadata and perform checks, (2) eliminates metadata copies among registers via modified register renaming, and (3) uses a dedicated metadata cache to reduce checking overhead. Furthermore, this paper extends Watchdog's mechanisms to detect bounds errors, thereby providing full hardware-enforced memory safety at low overheads.

1. Introduction

Languages such as C and C++ are the gold standard for implementing low-level systems software such as operating systems, virtual machine monitors, language runtimes, embedded software, and performance critical software of all kinds. Although safer languages exist (and are widely used in some domains), these low-level languages persist partly because they provide low-level control over memory layout and use explicit manual memory management (*e.g.*, `malloc` and `free`). Unfortunately, the unsafe attributes of these languages result in nefarious bugs that cause memory corruption, program crashes, and serious security vulnerabilities.

This paper focuses primarily on eliminating memory corruption and security vulnerabilities caused by one specific aspect of low-level languages: memory deallocation errors, which directly result in dangling pointer dereference bugs

```
Heap based                Stack based
int *p, *q, *r;           int* q;
p = malloc(8);            void foo() {
...                         int a;
q = p;                      q = &a;
...                       }
free(p);                  int main() {
r = malloc(8);              foo();
...                         ... = *q;
... = *q;                 }
```

Figure 1. Dangling pointer errors involving the heap and stack. On the left, freeing p causes q to become a dangling pointer. The memory pointed to by q could be reallocated by any subsequent call to `malloc()`. On the right, `foo()` assigns the address of stack-allocated variable a to global variable q. After `foo()` returns, its stack frame is popped, and q points to a stale region of the stack, which any intervening function call could alter. In both cases, dereferencing q can result in garbage values or data corruption.

and use-after-free (UAF) security vulnerabilities. In essence, we seek to enforce *safe manual memory management* and thereby eliminate an entire class of security vulnerabilities and uncaught memory corruption bugs. Figure 1 shows two use-after-free errors, including a dangling reference to a reallocated heap location (left) and a dangling pointer to the stack (right).

Use-after-free (UAF) errors have proven to be just as severe and exploitable as buffer overflow errors: they too potentially allow an attacker to corrupt values in memory [6], inject malicious code, and initiate return-to-libc attacks [31]. Use-after-free vulnerabilities have been used in real-world attacks, and many such vulnerabilities in mainstream software include `CVE-2010-0249` in Microsoft's Internet Explorer, `CVE-2009-1690` in Apple's Safari and Google's Chrome browsers, and `CVE-2011-2373` in Mozilla's Firefox browser. The number of use-after-free vulnerabilities reported has been increasing in recent years, in contrast with the decline in buffer overflow vulnerabilities reported [1].

The fundamental cause of use-after-free vulnerabilities is that programming languages providing manual memory management (such as C/C++) do not ensure that programmers use those features safely. Unfortunately, the commonly advocated alternative — developing software in garbage-

collected languages — is not always easily applicable in all domains (such as for operating systems or high-performance real-time code) where real-time response or low-level control over memory is required. Porting legacy code to these safer languages is not always cost effective, and using conservative garbage collection [4] is not always feasible due to pause times and observed memory leaks in long-running software [32].

Given their importance, there has been increased effort in detecting these use-after-free errors, but significant challenges remain. One approach is to randomize the location of heap objects [19, 28] to probabilistically transform dangling pointer dereferences into hard-to-exploit non-deterministic errors. Although useful for mitigating some use-after-free exploits, such approaches fail to detect use-after-free errors on the stack and suffer from locality-destroying memory fragmentation. Other approaches seek to detect errors by tracking the allocation/deallocation status of regions of memory (via shadow space in software [26], with hardware [5, 34], or page-granularity tracking via virtual memory mechanisms [10, 20]). Alternative approaches track and check unique identifiers either completely with software [3, 23, 29, 35] or with hardware acceleration [7]. In general, prior approaches generally suffer from one or more of the following issues: failure to detect all use-after-free errors (due to reallocation of memory), incompatibilities (due to memory layout changes), require significant changes to the tool chain, high memory overheads, or high performance overheads. Such overheads typically limit the use of software-only approaches to only debugging contexts rather than being used all the time in deployed code. Given that manual memory management in low-level languages like C is still ubiquitous, we seek a highly compatible means of eliminating use-after-free vulnerabilities completely with low enough overheads to be used all the time.

This paper proposes Watchdog, a hardware proposal that enforces safe and secure manual memory management even in the presence of reallocations. Watchdog builds upon prior proposals for identifier-based use-after-free checking [3, 7, 23, 29, 35] using disjoint metadata [23]. The Watchdog hardware performs the majority of the work, relying on the software runtime to inform it about memory allocations and deallocations. Watchdog associates a unique identifier with every memory allocation, and it marks the identifier as invalid on memory deallocations. Every pointer has an associated identifier that is propagated on pointer operations. For pointers resident in memory, these identifiers are maintained in a disjoint metadata space. Watchdog performs use-after-free checks to ascertain that the identifier associated with the pointer is still valid before every memory access. As identifiers are never reused (they are 64-bit unique values), Watchdog detects all use-after-free errors even in the presence of reallocations.

The Watchdog hardware employs several optimizations to reduce the performance penalties of identifier-based checking. First, Watchdog uses micro-op (μop) injection [8]

to perform metadata propagation and checking. Second, Watchdog uses an identifier metadata encoding that reduces the validity check to a single load-and-compare μop. Third, Watchdog identifies memory operations that load or store pointers conservatively in unmodified binaries (by assuming any 64-bit integer may contain a pointer) or more precisely using ISA-assisted identification, which allows the compiler to annotate loads and stores of pointers. Fourth, the hardware uses copy elimination via register remapping to eliminate metadata copies within the core. Performance measurements of a simulated x86 processor on twenty benchmarks indicate that the runtime overhead is 15% on an average for detecting all use-after-free errors.

Furthermore, this paper shows that the basic mechanisms in Watchdog are easily extended to include pointer-based bounds checking in hardware [9], further supporting the applicability and utility of the approach. Many of the same optimizations apply directly, resulting in a system with 24% average performance overhead that enforces full memory safety and thus provides comprehensive prevention of all use-after-free and buffer overflow vulnerabilities.

2. Background

This section provides background on techniques for detecting use-after-free errors that are most closely related to Watchdog; other approaches are discussed in Section 10. Although a long standing problem, there are no hardware proposals that detect all use-after-free errors (*i.e.*, even in the presence of reallocations) while maintaining memory layout compatibility with existing code.

Existing use-after-free checking approaches can be broadly classified into two categories based on the tracking and checking scheme used: (1) *location based* and (2) *identifier based*. As described in the subsections below, location-based approaches fail to detect use-after-free errors to re-allocated memory, and thus lack the comprehensive detection provided by identifier-based approaches. In contrast, identifier-based approaches typically use inline metadata, which introduces incompatibilities due to memory layout changes. Watchdog is the first hardware implementation of identifier-based checking with disjoint metadata, allowing it to avoid the incompatibility of inline metadata while retaining comprehensiveness.

2.1 Location-Based Checking

Location-based approaches (*e.g.*, [17, 25, 34]) use the location (address) of the object to determine whether it is allocated or not. An auxiliary data structure records the allocated/deallocated status of each location, and it is updated on memory allocations (*e.g.* `malloc()`) and deallocations (*e.g.* `free()`). On each memory access, these auxiliary structures are consulted to determine whether the dereferenced address is currently valid (allocated) memory. Common implementations of these object-location based approaches use either a splay tree [11, 17] or a shadow space [2, 25, 34] as the auxiliary data structure to provide efficient lookups.

These techniques are good debugging aids and detect many common errors. However, their inability to detect errors in the presence of reallocations fundamentally limits the efficacy of these approaches. Whenever a location is reallocated this approach erroneously allows the dereference of a dangling pointer reference to proceed. The information tracked by this approach is insufficient to determine that the pointer's original allocation area was freed and is now being used again for another potentially unrelated purpose.

2.2 Identifier-Based Checking

An alternative approach is the *allocation identifier* approach [3, 7, 23, 29, 35], which associates a unique identifier with each memory allocation. Whenever memory is deallocated (by a call to `free()` or popping a call stack frame), the unique identifier associated with that allocation is marked as invalid. On a memory access, the system checks that the unique allocation identifier associated with pointer used to make the access is still valid. Identifiers are never reused, and thus any memory access to a deallocated object will find that the identifier is no longer valid.

The per-allocation identifier must be tracked so that it can be checked on every memory access. Thus, each pointer has an associated identifier, and this identifier is propagated as the pointer is copied or pointer arithmetic is performed. This metadata is tracked both for pointers in registers and pointers in memory. In memory, this metadata is typically maintained inline (*i.e.*, a fat pointer) [3, 7, 29, 35]. Such inline metadata can introduce code incompatibilities due to memory layout changes. In addition, arbitrary casts in the program can corrupt inline metadata resulting in uncaught errors. To avoid the problems of inline metadata, a recently proposed software-only identifier-based approach employs disjoint metadata [23]. The key advantage of identifier based approaches is the ability to detect all dangling pointer errors. However, as they require a lookup on every memory access, they can have significant performance overheads.

2.3 Implementation Approaches and Comparison

Table 1 provides the comparison of various schemes under the common umbrella of location based and identifier based approaches. Use-after-free checking can be implemented: (1) by source-to-source translation, (2) during compilation, (3) via binary rewriting, (4) completely in hardware, or (5) various hybrids of these approaches. One advantage of source-to-source, compiler-based or hybrid instrumentation is that the identity of instructions manipulating pointers is known. Thus, identifier based approaches, which maintain identifier information with each pointer, can be more efficiently implemented on the source code or within the compiler. In contrast, binary translation and hardware-based approaches generally do not know a priori which instructions manipulate pointers. Hence, such approaches have primarily used location-based checking, resulting in the inability to detect errors in the presence of reallocations and thus limiting their effectiveness in preventing security vul-

	Approach	Instrument.	Runtime	Metadata	Casts	Compre.
Location	MC [26]	Binary	10x	Disjoint	Y	N
	JK [17]	Compiler	10x			
	LBA [5]	H/W	1.2x			
	SProc [13]	H/W	1.2x			
	MTrac [34]					
Identifier	SafeC [3]	Source	10x	Inline	N	Y
	P&F [29]	Source	5x	Split		
	MSCC [35]	Source	2x	Split		
	Chuang [7]	Hybrid	1.2x	Inline		
	CETS [23]	Compiler	2x	**Disjoint**	Y	Y
	Watchdog	**H/W**	**1.2x**			

Table 1. Comparison of representative location-based and identifier-based approaches with instrumentation method, runtime overhead, metadata organization, safety even with arbitrary type casts, and comprehensive (Compre.) error detection in the presence of memory reallocations.

nerabilities. Chuang *et al.* [7] proposed a compiler-based identifier-based checker with hardware acceleration and inline metadata. Its inline metadata changes the layout of objects in memory (which can cause code incompatibilities) and allows arbitrary type casts to corrupt metadata (which can compromise comprehensive detection).

3. Watchdog Approach

The goal of Watchdog is to enforce safe and secure manual memory management by providing *comprehensive detection*—detecting all memory management errors even in the presence of reallocations—and doing so while keeping the overheads of checking every memory access low enough to be widely deployed in live systems. To provide comprehensive detection, Watchdog employs identifier-based checking of use-after-free errors almost entirely in hardware, relying on the software runtime only to provide information about memory allocations and deallocations. To localize the hardware changes, this checking is implemented by augmenting instruction execution by injecting extra μops. Furthermore, Watchdog aims to provide source compatibility (*i.e.*, few source code changes) and binary compatibility (*i.e.*, library interfaces unchanged) by leaving the data layout unchanged using a disjoint shadow space for metadata.

3.1 Operation Overview

In Watchdog, the hardware is responsible for identifier propagation and checking. Once the memory allocations are identified with the help of a modified `malloc()` and `free()` runtime library, a unique identifier is provided to every memory allocation and associated with the pointer pointing to the allocated memory. As pointers can be resident in any register, conceptually Watchdog extends every register with a sidecar *identifier register*. Pointers can also reside in memory, so Watchdog provides a shadow memory that shadows every word of memory with an identifier for point-
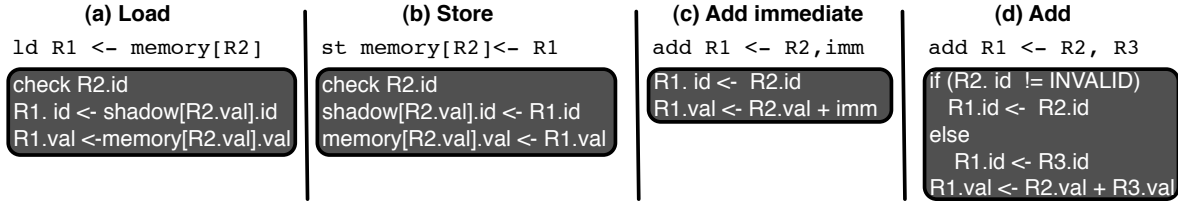
(a) Load	**(b) Store**	**(c) Add immediate**	**(d) Add**
`ld R1 <- memory[R2]`	`st memory[R2]<- R1`	`add R1 <- R2,imm`	`add R1 <- R2, R3`

check R2.id
R1. id <- shadow[R2.val].id
R1.val <-memory[R2.val].val
	check R2.id
	shadow[R2.val].id <- R1.id
	memory[R2.val].val <- R1.val
		R1. id <- R2.id
		R1.val <- R2.val + imm
			if (R2. id != INVALID)
			  R1.id <- R2.id
			else
			  R1.id <- R3.id
			R1.val <- R2.val + R3.val

Figure 2. Identifier (id) metadata checking and propagation through `load`, `store`, `add` immediate and `add`.

---

ers. To propagate and check the metadata, Watchdog injects μops in hardware. On memory deallocations, the identifier associated with the pointer pointing to the memory being deallocated is marked as invalid. On every memory access, Watchdog checks to ascertain if the identifier associated with the pointer being dereferenced is still valid. Accessing a memory location with an invalid identifier results in an exception. The following subsections explain each of these operations performed by Watchdog.

3.2 Checks on Memory Accesses

The hardware performs identifier validity checking by inserting a `check` μop before every memory access. The `check` μop uses the sidecar identifier metadata associated with the pointer register being dereferenced and checks whether the retrieved identifier is in the set of valid identifiers. A check failure triggers an exception, which can be handled by the operating system by aborting the program or by invoking some user-level exception handling mechanism. Performing an expensive check on every memory access could result in large performance overheads, so Section 4 describes the techniques we use to avoid expensive validity checks by reorganizing the identifier metadata and employing hardware caching.

3.3 In-Memory Pointer Metadata

As pointers can be resident in memory, the identifier metadata also needs to be maintained with pointers in memory. To maintain memory layout compatibility, the hardware maintains the per-pointer metadata in the shadow memory. Conceptually, every word in memory has identifier metadata in the shadow memory. When a pointer is read from memory, the metadata associated with the pointer being read is also read from the shadow memory. To implement this behavior (see Figure 2a), for every load instruction the Watchdog hardware injects (1) a `check` μop to perform the check, (2) a μop to perform the load of the actual value into the register and (3) a `shadow_load` μop to load the identifier (ID) metadata from the shadow memory space. Stores are handled analogously (also shown in Figure 2b). We assume pointers are word aligned (as is required by some ISAs and is generally true with modern compilers even for x86), which allows the the shadow load/store μops to accesses the shadow space via an aligned load/store in a single cache access.

The shadow space is placed in a dedicated region of the virtual address space that mirrors the normal data space.

Placing the shadow space into the program's virtual address space allows shadow accesses to be handled as normal memory accesses using the usual address translation and page allocation mechanisms of the operating system. Current 64-bit x86 systems support 48-bit virtual addresses, so the the hardware uses a few high-order bits from the available virtual address space to position the shadow space. This organization allows the shadow_load/shadow_store μop to convert an address to a shadowspace address via simple bit selection and concatenation.

Accessing the shadow space on every memory operation would result in significant performance penalties, so Section 5 describes the mechanisms we use to reduce the number of metadata accesses by inserting metadata load/store μops only for those memory operations that might actually load or store pointer values.

3.4 In-Register Metadata

To ensure that the checks inserted before a memory access have the correct identifier metadata, the identifiers must be propagated with all pointer operations acting on values in registers (pointer copies and pointer arithmetic). For example, when an offset is added or subtracted from a pointer, the destination register inherits the identifier of the original pointer. Figure 2 shows the identifier propagation with addition operations as a result of pointer arithmetic. Such register manipulation instructions are extremely common, so copying the metadata on each operation (say, via an inserted μop) would be extremely costly. Instead, Section 6 describes Watchdog's use of copy elimination via register renaming to reduce the number of propagation μops inserted.

3.5 Watchdog Usage Model

Watchdog would likely first be deployed on an opt-in basis similar to the deployment scenario with Microsoft's page-based no-execute Data Execution Prevention (DEP) feature. Initially, DEP was not enabled by default as it can break some programs that use self-modifying code or JIT-based code generation. Similarly, Watchdog would not be enabled for programs that might violate Watchdog's assumptions until after developers have explicitly tested their code with it. Like DEP, which is now enabled by default for most new software, we anticipate similar adoption for Watchdog.

This section has described the basic approach and has outlined three implementation optimizations to make it efficient: implementing efficient checks, identifying pointer ac-
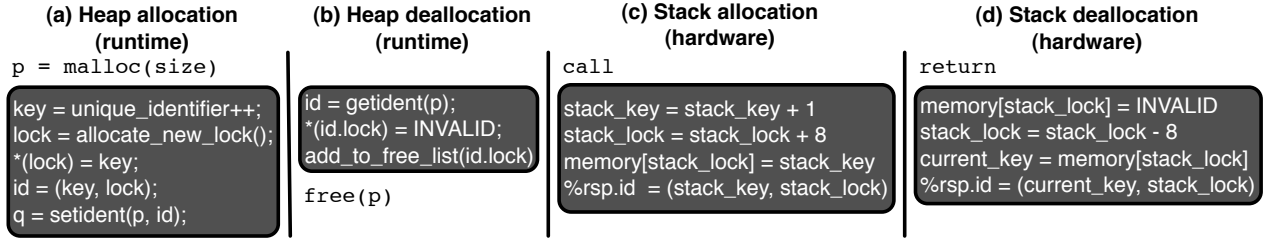
(a) Heap allocation (runtime)	**(b) Heap deallocation (runtime)**	**(c) Stack allocation (hardware)**	**(d) Stack deallocation (hardware)**
`p = malloc(size)`	`id = getident(p);`	`call`	`return`
key = unique_identifier++;	*(id.lock) = INVALID;	stack_key = stack_key + 1	memory[stack_lock] = INVALID
lock = allocate_new_lock();	add_to_free_list(id.lock)	stack_lock = stack_lock + 8	stack_lock = stack_lock - 8
*(lock) = key;		memory[stack_lock] = stack_key	current_key = memory[stack_lock]
id = (key, lock);	`free(p)`	%rsp.id = (stack_key, stack_lock)	%rsp.id = (current_key, stack_lock)
q = setident(p, id);			

Figure 3. Identifier (id) allocation and deallocation with malloc/free and call/return

cesses, and register renaming techniques to avoid unnecessary μops. The next three sections describe these design optimizations, respectively.

4. Efficient Use-After-Free Checking

Watchdog performs a check as part of every load and store. Memory operations are common, so this operation must be lightweight to achieve low performance overheads. This section describes: (1) the engineering of the identifiers to make lookups cheap and (2) using a hardware cache to further accelerate these checks.

4.1 "Lock and Key" Lookups

To accelerate checks, Watchdog uses a hardware implementation of the *lock and key* checking approach previously used by some software-only checkers [23, 29, 35]. The lock and key identifier technique transforms a validity check into just a load and comparison by splitting the identifier into two sub-components: a *key* (a 64-bit unsigned integer) and a lock (a 64-bit address which points to a location in memory). The memory location pointed to by the lock is called the *lock location*. The system maintains the invariant that if the identifier is valid, the value contained in the lock location is equal to the key's value. With a lock and key identifier, a check is a single memory access plus an equality comparison. This check works because: (1) the key is written into the lock location at allocation time, (2) the contents of the lock location is changed upon deallocation, and (3) keys are unique and thus no subsequent allocation will ever reset the lock location to value that would cause a spurious match (even if the underlying memory or the lock locations are subsequently reused).

Memory allocation/deallocation occurs when (1) the runtime performs such operations on the heap and (2) new stack frames are created/deleted on function entry and exits. Correspondingly, Watchdog allocates/deallocates identifiers on these operations. Heap allocation is fairly uncommon compared to function calls, so Watchdog relies on the runtime software to perform identifier management for the heap. In contrast, the Watchdog hardware performs the identifier management for function calls/returns.

On each heap memory allocation, the software runtime allocates both a unique 64-bit key and a new lock location from a list of free locations, and the runtime writes the key value into the lock location. The runtime conveys the identifier to the hardware using the `setident` instruction that takes two register inputs: (1) a pointer to the start of the memory being allocated and (2) the unique lock and key identifier (128-bit) being assigned as shown in Figure 3a. On memory deallocations, the runtime obtains the identifier associated with the pointer being freed using the `getident` instruction that takes the pointer being freed as the register input as shown in Figure 3b. The runtime then uses the identifier metadata to write an `INVALID` value to the lock location. The runtime then returns the lock location to the free list. To prevent double-frees and calling `free()` on memory not allocated with `malloc()`, the runtime also checks that the pointer's identifier is valid as part of the `free()` operation.

To perform identifier management for stack frames on calls and returns, the hardware injects μops to maintain an in-memory stack of lock locations whose top of stack is stored in a new `stack_lock` control register. The next key to be allocated is maintained with a separate `stack_key` control register. On a function call, the hardware injects four μops to: allocate a new key, push that key onto the in-memory lock location stack, and associate the new key and lock location with the stack pointer (see Figure 3c). On function return, the identifier associated with the stack pointer is restored to the identifier of the current stack frame. This operation is accomplished by reading the value of the key from the memory location pointed by the stack lock register after the stack manipulation (also 4 μops, as shown in Figure 3d).

Figure 4a illustrates the lock and key identifier scheme where two pointers p and q point to the different parts of the same object. Hence both p and q have the same ID in shadow memory with key being 5 and lock being address $0xB0$. The lock location at address $0xB0$ pointed by the lock part of the ID also has 5 as the key indicating that the object pointed by the pointers p and q are still allocated and valid.

Watchdog inserts a `check` μop before each memory access. This single μop reads the metadata from a register (which contains both the lock and key), loads the value currently at the lock location, and then compares it to the key. If the value loaded does not match the key, the memory associated with this pointer was previously deallocated, thus the access is invalid and the hardware raises an exception. Figure 4b illustrates the validity check that consists of a single memory access plus an equality comparison rather than some more expensive hash table or tree lookup operation.
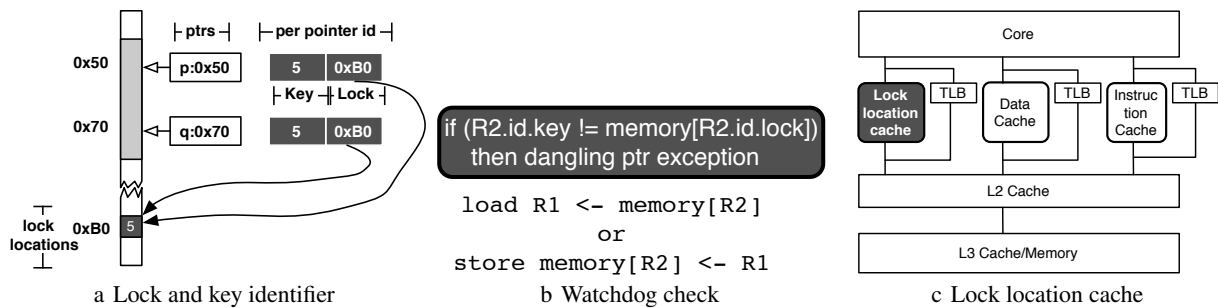
a Lock and key identifier   b Watchdog check   c Lock location cache

Figure 4. (a) Watchdog's lock and key identifier. Here two pointers *p* and *q* point to different part of the same object and hence have the same identifier. (b) the placement of the lock location cache (shaded). (c) Watchdog check before a load and store is a memory access and a equality comparison with lock and key identifier.

4.2 Lock Location Cache

Even with this efficient validity check, each `check` μop still performs a memory access, which increases the demand placed on the cache ports. To mitigate this impact, Watchdog optionally adds a *lock location cache* to the core, which is accessed by the `check` μop and is dedicated exclusively for lock locations. Just as splitting the instruction and data caches increases the effective cache bandwidth (by separating instruction fetches from loads/stores), this additional cache is used to provide more bandwidth for accessing lock locations. This cache becomes a peer with the instruction and data caches (as shown in Figure 4c), has its own (small) TLB, and uses the same tagging, block size, and state bits used to maintain coherence among the caches. Memory allocations and deallocations update lock location values, so these operations also access the lock location cache. Even a small lock location cache (*e.g.*, 4KB) can be effective because (1) lock locations (8 bytes per object currently allocated) are small relative to the average object size and (2) the lock locations region has little fragmentation and exhibits reasonable spatial locality because lock locations are reallocated using a LIFO free list. Cache misses are handled just like misses in the data cache.

5. Identifying Pointer Load/Store Operations

Watchdog conceptually maintains a lock and key identifier with every pointer in a register or memory. However, binaries for standard ISAs do not provide explicit information about which operations manipulate pointers. In the absence of such information, propagating metadata with every register and memory operation would require many extra memory operations, resulting in substantial performance degradation. This section describes two techniques for identifying pointer operations, the results of which can be used to reduce the number of accesses to the metadata space.

5.1 Conservative Pointer Identification

To enable Watchdog to work with reasonable overhead without significant changes to program binaries, we observe that,

for current ISAs and compilers, pointers are generally word-sized, aligned, and resident in integer registers. Based on this observation, Watchdog conservatively assumes that only a 64-bit load/store to an integer register may be a pointer operation, whereas floating point load/stores and sub-word memory accesses are non-pointer operations. Watchdog does not insert additional metadata manipulation μops for such non-pointer operations.[1] As evidence of the effectiveness of this heuristic, the left bars in Figure 5 show that this approach classifies 31% of memory operations as potentially loading/storing a pointer.

5.2 ISA-Assisted Pointer Identification

Although the above conservative heuristic is effective, we also explore more precise identification by extending the ISA with load and store variants that indicate whether a pointer is being loaded or stored. Using these annotations, the compiler, which generally knows which operations are manipulating pointers, is responsible for conservatively selecting the proper load/store variants. To experimentally study the potential benefits of this approach without performing significant modifications to a compiler backend, we used a profiling pass to determine which static instructions ever load or store valid pointer metadata. The profiling pass monitored every access to the metadata space and identified those static instructions that ever loaded valid metadata. For subsequent runs, we considered these static memory operations as having been marked by the compiler as load/stores of pointers.[2] The right bar of each benchmark in Figure 5 shows that this approach reduces the number of memory accesses classified as pointer operations to just 18% on average.

---

[1] One potentially problematic case is the manipulation of pointers using byte-by-byte copies (*e.g.* `memcpy()`). We found that compilers for x86 typically use word-granularity operations. In other cases, we have modified the standard library as needed.

[2] Although we see this approximation as primarily an experimental aide, such an approach might actually be useful for instrumenting libraries or other code that cannot be recompiled.
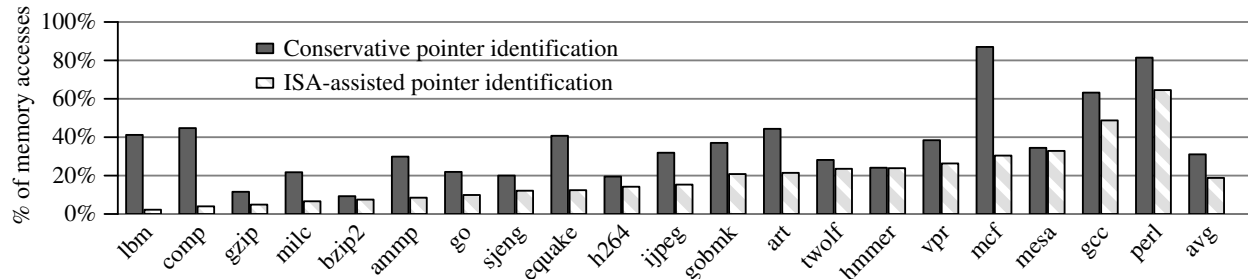
Figure 5. Percentage of memory accesses metadata for conservative and ISA-assisted identification.

6. Decoupled Register Metadata

As described thus far, Watchdog explicitly copies metadata along with each arithmetic operation in registers. This section briefly discusses and discards the straightforward approach of widening each register with additional metadata. Although such a design might be appropriate for an in-order processor core, our Watchdog implementation targets out-of-order dynamically scheduled cores. Thus, this section: (1) describes a decoupled metadata implementation of Watchdog in which the data and metadata are mapped to different physical registers within the core, (2) discusses what μops would be inserted to maintain the decoupled register metadata, and (3) shows how metadata propagation overheads can be reduced via previously-proposed copy elimination modifications to the register renaming logic.

6.1 Strawman: Monolithic Register Data/Metadata

As presented in Section 3, Watchdog views each register as being widened with a sidecar to contain identifier metadata. Although that design is conceptually straightforward—especially for an in-order core—it suffers from inefficiencies. First, every register write (and most register reads) must access the sidecar metadata, increasing the number of bits read and written to the register file; although not necessarily a performance problem, this could be a energy concern. Second, a more subtle issue is that treating register data/metadata as monolithic causes operations that write just the data or metadata to become partial register writes. These partial register accesses introduce unnecessary dependencies between μops, for example, the load μop and the metadata load μop. These serializations can have several detrimental effects, including (1) increasing the load-to-use penalty of pointer loads, (2) stalling subsequent instructions if either of the loads miss in the cache, and (3) limiting the memory-level parallelism by serializing the load and the metadata load. In our initial experiments with various implementations of monolithic registers, we found the performance impact of such serializations to be significant.

6.2 Decoupled Register Data/Metadata

To address these performance issues, Watchdog decouples the register metadata by maintaining the data and metadata in separate physical registers. Each architectural register is

mapped to two physical registers: one for data and one for the metadata. With this change, individual μops generally operate on either the data or the metadata, removing the serialization caused by partial register writes of monolithic registers. Once decoupled, the metadata propagation and checking is almost entirely removed from the critical path of the program's dataflow graph.

With decoupled metadata, there are multiple cases for which register metadata must be propagated or updated. First, instructions such as adding an immediate value to a register simply copy the metadata from the input register to the output register. Second, some instructions never generate valid pointers (*e.g.*, the output of a sub-word operation or a divide is not a valid pointer), thus such instructions always set the metadata of the output register to be invalid. Third, either of the registers might be a pointer, so for such instructions Watchdog inserts a `select` μop, which selects the metadata from whichever register has valid metadata.[3]

Watchdog performs metadata propagation by changing the register renaming to reduce the number of extra μops inserted. In only one of these three cases described above does Watchdog actually insert μops; in the other cases (copying the metadata or setting it to invalid), Watchdog uses previously proposed modifications to register renaming logic [18, 30] to handle these operations completely in the register rename stage. Watchdog extends the maptable to maintain two mappings for each logical register: the regular mapping and a metadata mapping. Instructions that unambiguously copy the metadata (such as "add immediate", which has a single register input) update the metadata mapping of the destination register in the maptable with the metadata mapping entry of the input register. This implementation eliminates the register copies by physical register sharing, as there is a single copy of the metadata in a physical register [30]. To ensure that this physical register is not freed until all the mappings in the maptable are overwritten, these physical registers need to be reference counted. We adopt previously proposed techniques to efficiently implement reference counted physical registers [33]. Figure 6 shows how decoupled metadata avoids copy μops while avoiding the introduction of unnecessary dependencies between μops.

---

[3] If the ISA was further extended to allow the compiler to annotate such instructions, these select μops could also be eliminated.

Program with Watchdog uops	Map-Table	Renamed Instructions
A: check r2.id	r1:(p1, -),r2:(p2,p6),r3:(p3, -)	check p6
B: ld r1.id <- shadow[r2.val]	r1:(p1,**p7**),r2:(p2,p6),r3:(p3, -)	ld p7 <- shadow[p2]
C: ld r1 <- memory[r2]	r1:(**p4**,p7),r2:(p2,p6),r3:(p3, -)	ld p4 <- memory[p2]
D: add r3<- r1, 4	r1:(p4,p7),r2:(p2,p6),r3:(**p5,p7**)	add p5<- p4, 4
E: check r2.id	r1:(p4,p7),r2:(p2,p6),r3:(p5,p7)	check p6
F: st shadow[r2.val] <- r3.id	r1:(p4,p7),r2:(p2,p6),r3:(p5,p7)	st shadow[p2] <- p7
G: st memory[r2] <- r3	r1:(p4,p7),r2:(p2,p6),r3:(p5,p7)	st memory[p2] <- p5

Figure 6. Example illustrating register renaming with Watchdog μops and extensions to the map table. Watchdog inserted μops are shaded. The map table is represented by a tuple for each register. r:(a,b) means logical register *r* maps to physical register *a* according to the regular map table mapping and the logical register *r* maps to a 128-bit physical register *b* according to the Watchdog mapping. Watchdog introduced load and store μops access the shadow memory (`shadow`) for accessing the metadata. The watchdog mapping of $-$ indicates the invalid mapping (the register currently contains a non-pointer value).

7. Implementation Considerations

Custom memory allocation. For programs that use custom memory allocators (*e.g.*, by requesting a region of memory which it then partitions), by default Watchdog will check the allocation status of the entire region of memory. However, if the the programmer instruments the custom memory allocator, Watchdog will then be able to perform exact checking for these allocators.

Global variables and initialization. Global variables are never deallocated, so all pointers to globals (data segment) are given the same single global identifier, which ensures that the validity check always passes. One way to obtain the address of a global variable is by PC-relative addressing modes. Watchdog handles this case by associating the global identifier with any address generated by PC-relative addressing modes. A related issue is initializing the metadata space for the global variables. C programs are allowed to initialize non-null pointers in the global space by initializing such pointers to point to other objects in the global space. To properly set the metadata to support initialized global pointers, Watchdog also initializes the entire metadata shadow space for the global data segment with the global identifier.

Multithreading. Although we do not evaluate Watchdog in the context of multithreaded programs, Watchdog would ideally seamlessly support multithreaded workloads (an issue not typically addressed by existing software-only approaches). As with software-only approaches, if operations on pointers occur non-atomically, interleaved execution and data races (either unintentional races or intentional races used in lock-free concurrent data structures) can cause errors. Watchdog will work seamlessly with multithreaded workloads if its implementation ensures: (1) the runtime has a thread-safe way to allocate unique identifiers (easily handled with a thread-local variable and partitioning the space of identifiers), (2) pointer load and store's data and metadata accesses execute atomically, and (3) the check and memory operation occur atomically. Such atomicity could be provided by supporting a two-location atomic update operation in hardware. Alternatively, if we consider only lock-based data-race free programs, #2 above is not necessary and requirement #3 above can be relaxed if the check is moved to after the memory access and the system can tolerate a single invalid access before the exception is raised.

8. Integrating Bounds Checking

Although the focus of this paper is on use-after-free violations, Watchdog's overall approach and implementation was explicitly designed to mesh well with pointer-based bounds checking [3, 9, 15, 22, 24, 29, 35], which track base and bound metadata with pointers for precise byte-granularity bounds checking of all memory accesses. The metadata propagation and checking machinery proposed in this paper can be extended to provide efficient bounds checking — and thus hardware-enforced full memory safety.

To implement a fully hardware-enforced pointer-based memory bounds checker [9], we extend the Watchdog hardware in three ways. First, we extend the in-memory and register metadata with 64-bit base and 64-bit bound metadata (for a total of 256 bits of metadata per pointer). Second, we correspondingly widened the memory access μops injected for accessing in-memory metadata. Third, we extend checking to perform a range check based on the metadata by either: (1) injecting a new additional bounds check μop for each memory operation or (2) by extending the current check μop to perform both checks in parallel with one μop. As the bounds check consists of just two inequality comparisons (and requires no additional memory accesses), either implementation is likely feasible. We evaluate both alternatives. Note that Watchdog's mechanisms such as μop injection for metadata propagation/checking, pointer identification, decoupled side-car registers, and copy elimination using register renaming techniques all apply directly, which highlights that the utility of such techniques extends beyond just use-after-free checking.

To enforce bounds precisely, the hardware relies on byte-granularity bounds information provided by the compiler and/or runtime whenever a pointer is created [9]. For heap allocated objects, the `malloc` runtime library can convey such bounds information. For pointers to a stack-allocated

	Clock	3.2 GHz
Front-end	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.
	LQ size	64-entry LQ
	SQ size	36-entry SQ

|	L1 I$	32KB. 4-way, 64B blocks. 3 cycles
Memory Hierarchy	Prefetcher	2-streams, 4 blocks each
	L1 D$	32KB, 8-way, 64B blocks, 3 cycles
	Prefetcher	4-streams, 4 blocks each
	L1 ↔ L2 bus	32-bytes/cycle. 1 cycle.
	Private L2$	256KB, 8-way, 64B blocks, 10 cycles.
	Prefetcher	8 streams. 16 blocks.
	L2 ↔ 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
	Lock Location $	4KB, 8-way, 64B blocks

Table 2. Simulated processor configurations
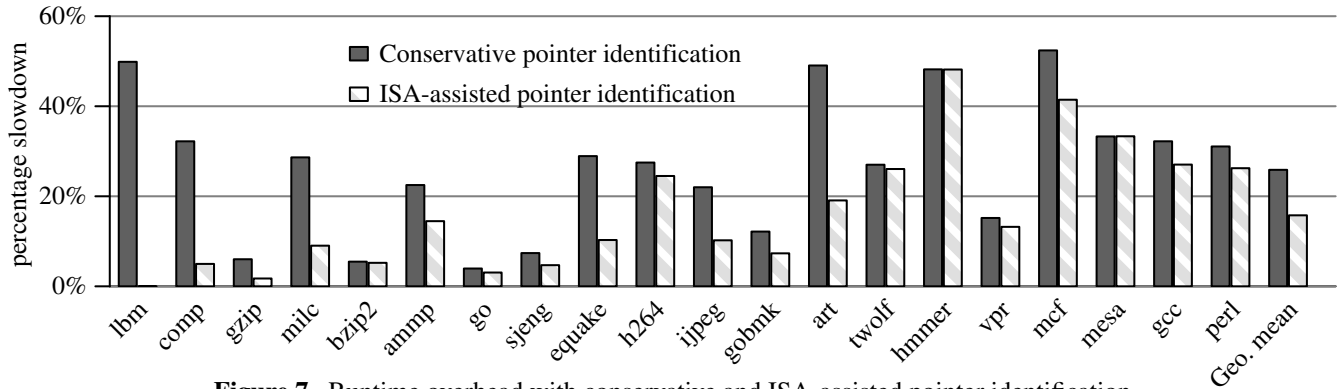
Figure 7. Runtime overhead with conservative and ISA-assisted pointer identification

or global object, precise checking requires the compiler to insert instructions to convey bounds information at pointer creation points. In the absence of such exact information, the hardware can still perform bounds checking — just less precisely — by restricting the bounds of pointers pointing to stack variables and globals to the range of the current stack frame and the global segment, respectively.

9. Experiments

This section provides an experimental evaluation of Watchdog and highlights: (1) its effectiveness in preventing security exploits, (2) its low performance overheads, and (3) its synergistic integration with bounds checking to provide full hardware-enforced memory safety.

9.1 Methodology

We used an x86-64 simulator that executes the user-level portions of statically linked 64-bit x86 programs. The simulator decodes x86 macro instructions and cracks them into a RISC-style μops. The out-of-order processor parameters (see Table 2) were selected to be similar to Intel's Core i7 "Sandy Bridge" processor. We modified the standard DL-malloc memory allocator to use the new instruction to inform the hardware of memory allocations and deallocations. We used twenty C SPEC benchmarks, including an enhanced version of the `equake` benchmark

that uses a proper multidimensional array and thus improves its baseline performance by 60%. We compiled the benchmarks using the GNU C compiler version 4.4 using standard optimization flags. We generally used the reference inputs, but used train/test inputs in some cases to ensure reasonable simulation times. We used 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.

9.2 Efficacy in Preventing Security Vulnerabilities

To evaluate the effectiveness of Watchdog in preventing use-after-free security exploits, we ran the 291 test cases for use-after-free vulnerabilities (CWE-416 and CWE-562) from the NIST Juliet Test Suite for C/C++ [27], which are modeled after various use-after-free errors reported in the wild. It successfully detected and thwarted the attack in all the 291 test cases, and it did so without any false positives.

9.3 Runtime Overheads of Use-after-Free Checking

Figure 7 presents the percentage execution time overhead of Watchdog over a baseline without any Watchdog instrumentation (smaller bars are better as they represent lower runtime overheads). The graphs contains a pair of bars for each benchmark. The height of the left and right bars represent the overhead of Watchdog with conservative pointer identification (25% on average) and ISA-assisted pointer iden-
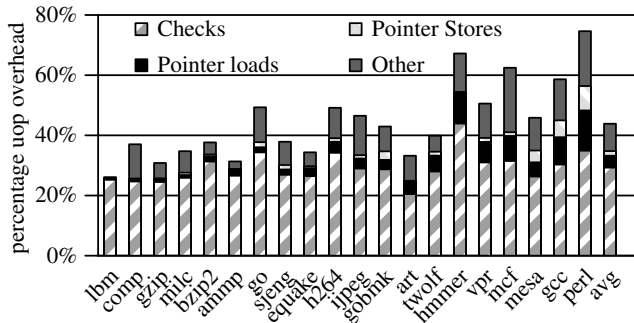
Figure 8. μop overhead

Figure 9. Performance with lock location cache

Figure 10. Memory overhead with words and pages

tification (15% on average), respectively. More than half of the benchmarks have runtime overheads less than 5% with ISA-assisted pointer identification. These runtime are substantially lower than the 50-133% overhead reported by related software-only schemes [23, 35]. More importantly, these overheads are likely low enough for use in production systems — critical for preventing security vulnerabilities — rather than just for debugging.

Watchdog performs its functionality by inserting μops, so the total number of μops inserted is instructive in understanding the sources of execution time overheads. Figure 8 presents the μop overhead when employing ISA-assisted pointer identification. The total height of the each bar represents the total μop overhead for the benchmark and each bar is divided into four segments: (1) checks, (2) pointer loads, (3) pointer stores, and (4) the μops to perform memory allocation/deallocation and identifier propagation in registers. On average, Watchdog executes 44% more μops than the baseline. The execution time overhead is lower than the μop overhead because these μops are off the critical path and thus execute in parallel as part of superscalar execution. The check μops account for bulk of the μop overhead (29% on average). Pointer metadata load and store μops account for 4% and 2% of the extra μops on an average but can be as high as 14% and 8%, respectively. The μop overhead due to propagation μops and memory allocation/deallocation operations (on the heap and the stack) account for the remaining μops (9% on average).

One potential source of performance overhead is the additional cache pressure due to the per-pointer shadowspace metadata. To isolate this effect, we performed a set of simulations configured to idealize the shadow memory accesses (metadata accesses occupy cache ports but never cache miss and to not actually consume space in the data cache). Making the metadata free of cache effects in this way changed the runtime overhead by only 4% on average (decrease from 15% to 11%), indicating that cache pressure effects are generally not dominant in these benchmarks.

To decrease contention on limited cache ports, the results presented thus far include a 4KB lock location cache (Section 4.2). Figure 9 reports the execution time overhead without this cache, in which all check operations use the limited data load ports. Without the lock location cache,
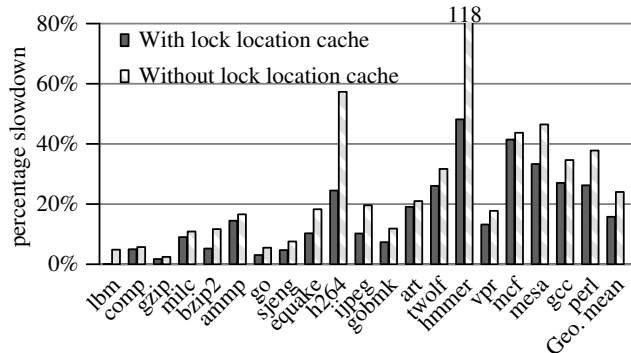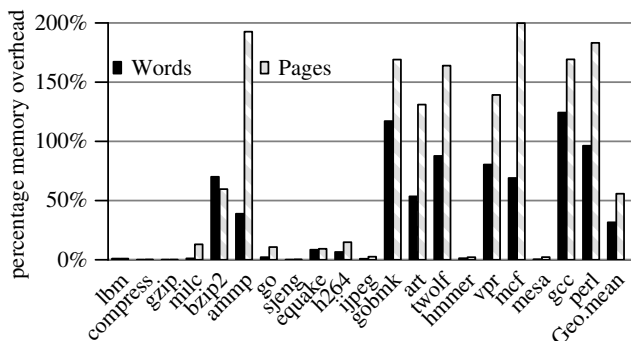
the overhead of Watchdog increases to 24% on average (up from 15%). The improvements are especially significant for benchmarks such as `hmmer` and `h264`, as these benchmarks already have high IPC and frequent memory accesses, both of which lead to significant contention for the data cache load ports in the absence of a lock location cache. These results are not particularly sensitive to the exact size of the lock location cache; for a 4KB cache, the miss rate is less than 1 miss per 1000 instructions for seventeen of the twenty benchmarks.

Figure 10 presents the memory overheads with ISA-assisted pointer identification over a baseline without any instrumentation. The memory overhead is negligible for the majority of the benchmarks. However, several of the benchmarks approach worst-case overheads of two shadow pages for each non-shadow page. The memory overhead is calculated in two ways: total words of memory accessed (left bar) and total 4KB pages of memory accessed (right bar), which reflects on-demand allocation of shadow space pages by the operating system. On an average, the memory overhead calculated these two ways is 32% and 56%, respectively. The difference in these metrics reflect the impact of fragmentation caused by page-granularity allocation of the shadow space. These memory overheads compare favorably to the overheads reported for garbage collection [14] or prior best-effort approaches for mitigating use-after-free errors such as heap randomization [20] and object-per-page approaches [10].
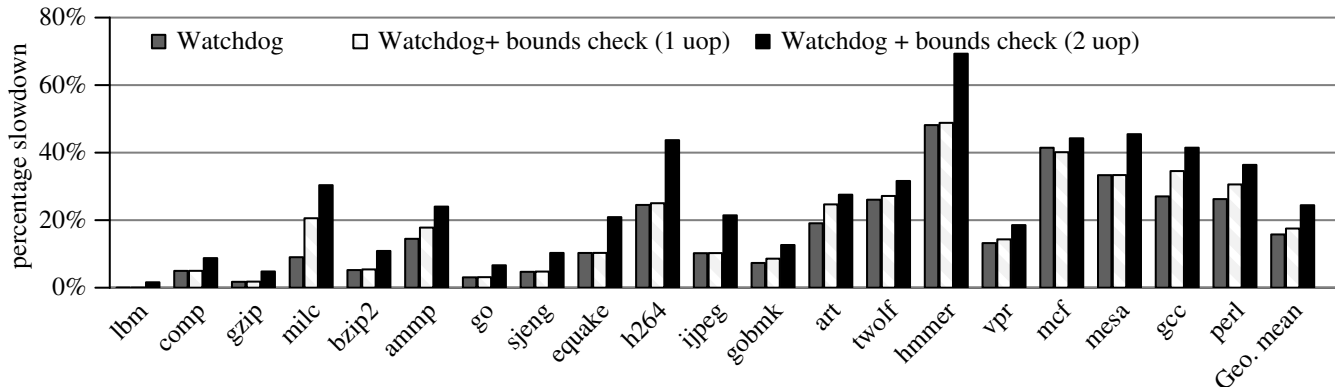
Figure 11. Runtime overhead with bounds checking with one or two check μops

9.4 Integrating Bounds Checking

Beyond use-after-free checking, Section 8 described how the proposed hardware can be extended to also perform bounds checking, thus enforcing full memory safety. Figure 11 presents the execution time overhead with ISA-assisted pointer identification for Watchdog extended to perform bounds checking by either injecting: (1) a single check μop (which performs both the bound and identifier checking) or (2) a pair of μops (one μop each for bounds and identifier checking). When implemented as a single μop, the overall number of μops is unchanged and results in only a 3% increase in execution time overhead (from 15% to 18%), primarily due to the cache pressure introduced by the larger in-memory metadata. When the bound check is implemented as an additional μop injection, the overhead increases by 6% to a total average overhead over the unmodified baseline of 24% for complete memory safety.

10. Related Work

Garbage collection (GC) is an alternative approach to eliminating dangling pointers for application domains where garbage collection is suitable. When combined with "heapification" [24] of escaping stack objects, garbage collection can eliminate all dangling pointers. Further, hardware support to either perform GC completely in hardware [21] or to flexibly accelerate GC have been proposed [16] to reduce the overheads. An alternative approach for enforcing safe manual memory management (at the cost of some modifications to the program source code) is to assert that an object's reference count is zero at deallocation time [12].

Several other approaches have been used to mitigate use-after-free errors. Probabilistic approaches [19, 28] use a randomized memory manager to allocate objects randomly on the heap and recycle the memory randomly, thereby making it difficult to exploit the errors. These techniques can leverage virtual address translation mechanism to detect errors without inserting any checking code. The open-source tools Electric Fence, PageHeap, and DUMA allocate each object on a different physical and virtual page. Upon deallocation, the access permission on individual virtual pages are disabled, increasing memory usage and causing high runtime overheads for allocation-intensive programs [10]. These schemes allocate one object per virtual and physical page, resulting in large memory overheads for programs with many small objects. This overhead can be partly mitigated by placing multiple objects per physical page but mapping a different virtual page for each object to the shared page [10, 20].

11. Conclusion

This paper described Watchdog, a hardware approach for safe and secure manual memory management via low-overhead, comprehensive use-after-free checking with disjoint metadata. To efficiently implement Watchdog, we used ideas from existing software-only approaches, explored ISA support for explicitly identifying pointer operations, described efficient decoupled register metadata via register renaming copy elimination, and utilized a lock location cache to accelerate checks. The resulting overhead is likely low enough to use in production (and not just debugging). Furthermore, we extended the mechanisms to perform bounds checking. The resulting system ensures full memory safety and provides comprehensive protection from use-after-free and buffer overflow vulnerabilities for a 24% performance penalty.

References

[1] National Vulnerability Database. NIST. `http://web.nvd. nist.gov/`.

[2] 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.

[3] 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.

[4] H.-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 197–206, June 1993.

[5] 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*, pages 377–388, June 2008.

[6] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks are Realistic Threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2005.

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

[8] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[9] 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.

[10] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, June 2006.

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

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

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

[14] M. Hertz and E. D. Berger. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. 2005.

[15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[16] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 418–428, June 2009.

[17] 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.

[18] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1998.

[19] 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*, pages 207–218, Oct. 2006.

[20] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: Trading Address Space for Reliability and Security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–124, Mar. 2008.

[21] M. Meyer. A Novel Processor Architecture with Tag-Free Pointers. In *IEEE Micro*, 2004.

[22] 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.

[23] 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.

[24] 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.

[25] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 65–74, 2007.

[26] 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*, pages 89–100, June 2007.

[27] *NIST Juliet Test Suite for C/C++*. NIST, 2010. http://samate.nist.gov/SRD/testCases/suites/Juliet-2010-12.c.cpp.zip.

[28] G. Novark and E. D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 573–584, 2010.

[29] 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.

[30] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

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

[32] J. Rafkind, A. Wick, M. Flatt, and J. Regehr. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management*, June 2009.

[33] A. Roth. Physical Register Reference Counting. *IEEE TCCA Computer Architecture Letters*, 7(1), Jan. 2008.

[34] 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*, pages 273–284, Feb. 2007.

[35] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 117–126, 2004.