HARDWARE-ENFORCED COMPREHENSIVE MEMORY SAFETY

THE WATCHDOG APPROACH EFFICIENTLY ELIMINATES AN ENTIRE CLASS OF SECURITY VULNERABILITIES BY ENFORCING MEMORY SAFETY IN HARDWARE. WATCHDOG MAINTAINS PER-POINTER BOUNDS AND IDENTIFIER METADATA IN A DISJOINT SHADOW SPACE TO ENSURE COMPATIBILITY WITH EXISTING CODE. CHECKING THIS METADATA ON EACH POINTER DEREFERENCE PROVIDES COMPREHENSIVE PROTECTION FROM BUFFER-OVERFLOW AND USE-AFTER-FREE ERRORS. THE WATCHDOG DESIGN AIMS TO REDUCE OVERHEAD WITHOUT INVASIVE HARDWARE CHANGES.

Santosh Nagarakatte
Rutgers University
Milo M. K. Martin
Steve Zdancewic
University of Pennsylvania

Low-level systems software—such as operating systems, virtual machines, language runtimes, embedded software, and performance-critical applications—is commonly written in unsafe languages, notably C and C++. These low-level languages remain prevalent because they provide high performance, direct access to the underlying hardware, and explicit control over memory management. Moreover, because such systems often consist of millions of lines of code, transitioning the computing ecosystem away from C and its variants is not feasible any time soon.

Unfortunately, C and its variants do not enforce memory safety. Informally, memory safety requires that all memory accesses performed by a program adhere to the language specification (that is, all accesses refer only to allocated memory within the prescribed object bounds). Violations of memory safety arise in two ways. First, a *spatial memory safety violation*, or buffer overflow, occurs when a program accesses a memory location outside of the allocated region of an object or array. Second, a *temporal memory safety violation* (also called a dangling pointer dereference or use-after-free error) occurs when a program accesses a memory location that has already been deallocated. See Figure 1 for examples of spatial and temporal safety violations.

Without memory safety, seemingly benign program bugs anywhere in the code base can cause silent memory corruption, difficult-to-diagnose crashes, and incorrect results. Worse yet, lack of memory safety enforcement is the root cause of a multitude of security vulnerabilities because it allows attackers to exploit a memory safety error to corrupt program execution by giving the vulnerable program a suitably crafted input. Buffer overflows, use-after-free errors, and other low-level vulnerabilities stemming from the lack of memory safety compromise the security of the computing ecosystem as a whole.[1,2]

Over the last few years, we've embarked on a research project focused on bringing memory safety enforcement to C and C++.[3-6] The goal is to retrofit unmodified C and C++ code with memory safety so that the code is as safe as code written in

Published by the IEEE Computer Society

managed languages (such as Java or C#) without sacrificing performance or low-level features such as explicit manual memory management. Focusing on comprehensive enforcement of memory safety addresses the root cause of memory-safety based security vulnerabilities and eliminates this entire class of commonly exploited security vulnerabilities rather than treating their symptoms.

This article describes Watchdog, a hardware-based proposal for enforcing comprehensive memory safety that leverages techniques we have used in our prior work on software-only enforcement of memory safety.[5,6] Our goal is to bring the highly compatible and comprehensive safety of the software-based approaches into the hardware domain to reduce performance overheads while limiting the design impact of the hardware changes.

The Watchdog approach

Watchdog adopts a pointer-based approach in which metadata is maintained with pointers, both in registers and in memory.[5-12] This per-pointer metadata is propagated through pointer manipulations such as pointer arithmetic. Whenever a pointer is loaded or stored from memory, its metadata is explicitly loaded and stored from a disjoint metadata space. Using a disjoint shadow space for the metadata leaves the data layout unchanged, which facilitates source compatibility (few source code changes) and binary compatibility (unchanged library interfaces). Intel recently released an update to its production compiler that includes a pointer checker based on disjoint per-pointer metadata, which further supports the feasibility of the general approach employed by Watchdog.[13]

Watchdog implements pointer-based checking by performing all the checking and propagation entirely in hardware. Watchdog relies on the software runtime only to provide information about memory allocations and deallocations. To prevent buffer overflow errors, the hardware maintains base and bounds metadata with every pointer and performs a range check on each memory access. To prevent use-after-free vulnerabilities, Watchdog associates a unique identifier with each object and tracks

Spatial error

```
int *p, *q, *r;
void foo(int i) {
          p = malloc(8);
          ...
          q = p + i;
          ...
          *q = ...;
          ...
}
```

(a)

Heap-based temporal error

```
int *p, *q, *r;
p = malloc(8);
...
q = p;
...
free(p);
r = malloc(8);
...
... = *q;
```

(b)

Stack-based temporal error

```
int* q;
void foo() {
          int a;
          q = &a;
}
int main() {
          foo();
          ... = *q;
}
```

(c)

Figure 1. Spatial and temporal safety violation examples. In (a), pointer `q` is out-of-bounds and its dereference causes a spatial safety error. Temporal errors (use-after-free errors) can occur on both the heap (b) and stack (c). In both cases, dereferencing `q` can result in garbage values or data corruption. In (b), freeing `p` causes `q` to become a dangling pointer. The memory pointed to by `q` could be reallocated by any subsequent call to `malloc`. In (c), `foo()` assigns the address of stack-allocated variable `a` to global variable `q`. After `foo()` returns, its stack frame is popped, thereby `q` points to a stale region of the stack, which any intervening function call could alter.

this object identifier metadata with each pointer to the object (in addition to the base and bounds metadata). The system also tracks which identifiers are marked "valid" and marks an identifier "invalid" when the memory that underlies the object is deallocated. Prior to each memory access, the hardware performs a temporal safety check to ensure the identifier is still valid. Identifiers are never reused, so they are unique. Unique identifiers provide comprehensive detection of temporal safety violations, even when the memory underlying the object has been reallocated.

To improve the performance of pointer-based checking while minimizing the invasiveness of the proposed hardware changes, Watchdog employs several techniques. First, Watchdog localizes the hardware changes by implementing checking by injecting extra micro-operations (μops).[14] Second, Watchdog uses a "lock and key" encoding of the identifier metadata to reduce the validity check to a single load-and-compare μop.[6,12] Third, Watchdog identifies memory operations that load or store pointers either conservatively in unmodified binaries by assuming any 64-bit integer may contain a pointer, or more precisely by extending the instruction-set architecture (ISA) to allow the compiler to annotate loads and stores of pointers. Fourth, the hardware applies copy elimination via register remapping to eliminate metadata copies within the core.[15]

Metadata and checking

To enforce bounds precisely, the hardware relies on byte-granularity bounds metadata (a 64-bit base and a 64-bit bound) for each pointer, which is provided by the compiler or runtime by use of a special instruction that associates bounds information with a newly created pointer.[3] The hardware then uses the per-pointer metadata to verify that all memory accesses are within the pointer's base and bounds. For heap allocated objects, the `malloc` runtime library is extended to include such instructions. For a pointer to a stack-allocated or global object, precise checking requires the compiler to insert instructions to convey bounds information when the pointer is created. In the absence of such exact information, the hardware performs less-precise bounds checking by restricting the bounds of pointers pointing to stack and global variables to the range of the current stack frame and the global segment, respectively.

To enforce temporal safety, Watchdog associates a unique identifier with each memory allocation, checks each memory access to ensure that the identifier is still valid, and marks the identifier as invalid when the object is deallocated. A software data structure such as a hash table or tree could be used to check whether an identifier associated with the pointer is still valid, but performing such lookups on every memory access introduces significant performance overhead.[7] To meet our goal of low overhead, we split the unique identifier into two subcomponents to accelerate lookups: a *key* (a 64-bit unsigned integer) and a *lock* (a 64-bit address that points to a location in memory).[6,12,13] The memory location pointed to by the lock is called the *lock location*. The lock provides an efficient mechanism for determining whether the memory allocated for the pointer has been deallocated by maintaining the invariant that if the identifier is valid, the value stored in the lock location is equal to the key's value. When an object is deallocated, the runtime system "changes the lock" on the object by setting the lock location to an invalid value, which prevents pointers with the now-stale key from accessing the memory (by analogy, this is similar to how a landlord might change the lock on a door after a tenant moves out to deny future access). Figure 2a shows the lock and key metadata maintained by each pointer to provide comprehensive memory safety.

The Watchdog software runtime allocates both a unique 64-bit key (by incrementing a counter) and a new 64-bit lock location (from a free list of locations) on each heap memory allocation, 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: a pointer to the start of the memory being allocated, and the 256-bit metadata (64-bit base, 64-bit bound, and 128-bit unique lock and key identifier) that is being assigned as shown in Figure 3a. On memory deallocations,

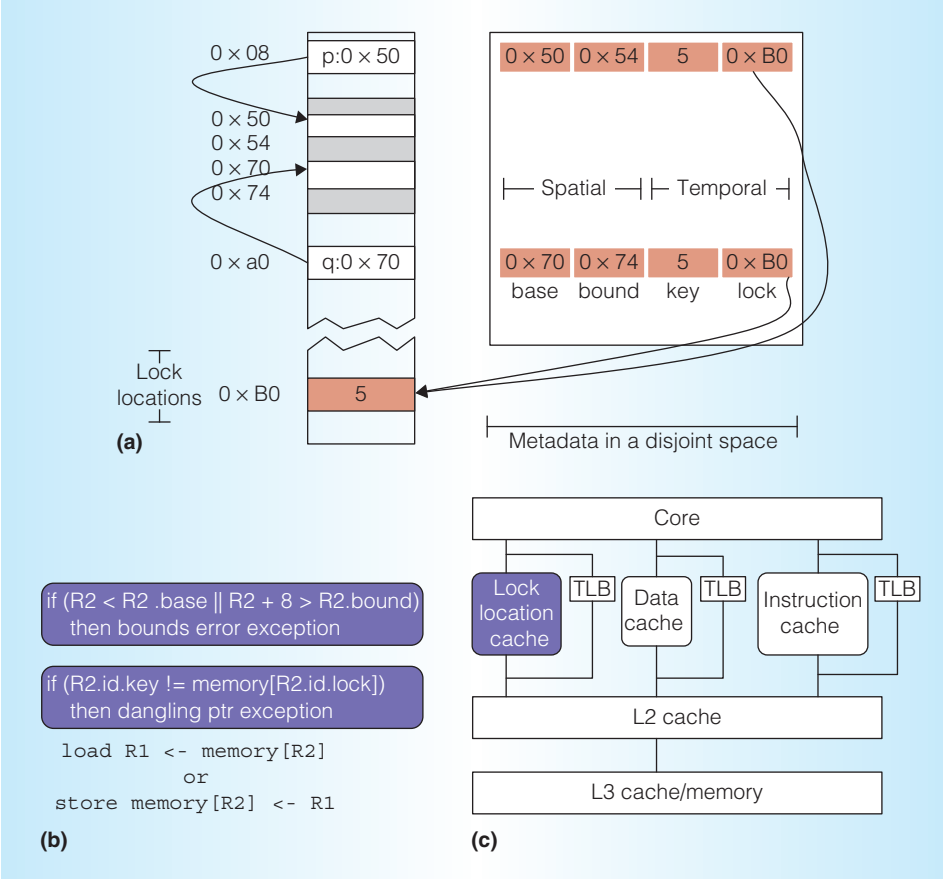

Figure 2. Per-pointer metadata: base, bounds, and unique identifier (ID) metadata organized as *key* and *lock*. Two pointers *p* and *q* point to different parts of the same object and hence have the same identifier but different bounds (a). Bounds and temporal safety check: the temporal safety check before a load and store is a memory access and an equality comparison with lock and key identifiers (b). Placement of the lock location cache (shaded) (c).

the runtime obtains the identifier associated with the pointer that is being freed using the `getident` instruction, which 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 and 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.

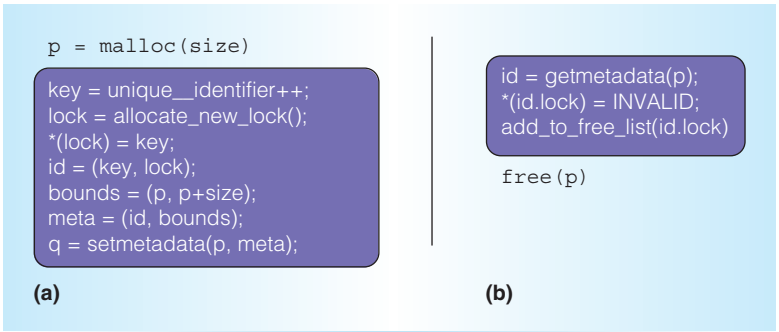

Figure 3. Metadata allocation and deallocation by the runtime with `malloc/free` and interfacing with the hardware using `setident` and `getident` instructions. The runtime creates a new key, allocates an available lock location, and assigns it to the pointer using the `setident` instruction on memory allocation (a). The key value at the lock location is changed to an INVALID value on memory deallocation (b).

In-register metadata

To track metadata for pointers resident in registers, Watchdog extends every register with a sidecar metadata register. The bounds and identifier metadata are propagated with

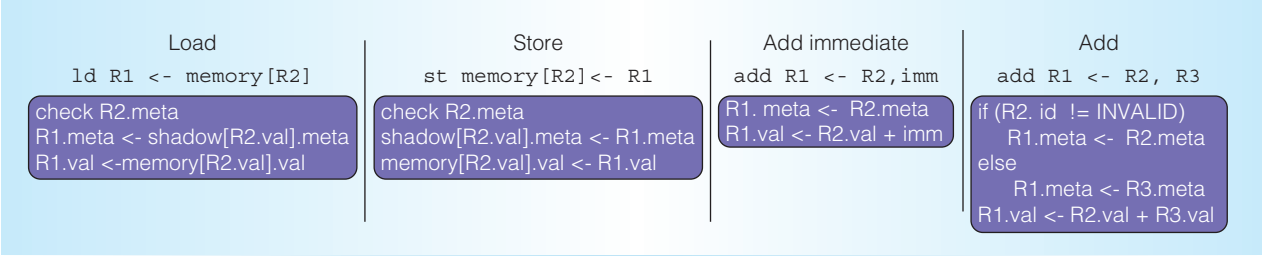

Figure 4. Metadata checking and propagation through *load*, *store*, *add-immediate*, and *add*. Metadata propagation occurs by accessing shadow memory with load (a) and store (b) instructions that load or store a pointer. Metadata propagation occurs in registers with add operations: when the instruction has only one operand (c), and when the instruction has two operands (d).

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 original pointer's metadata. Figure 4 shows the bounds and identifier metadata propagation with additional operations as a result of pointer arithmetic. Watchdog conceptually performs such metadata propagation by injecting extra propagation µops (see Figure 4). Such register manipulation instructions are common, so copying the metadata on each operation (say, via an inserted µop) would be costly. Instead, Watchdog uses copy elimination via register renaming to reduce the number of propagation µops inserted.

In-memory pointer metadata

Pointers can also reside in memory, so Watchdog also maintains the metadata with pointers in memory. The hardware maintains the per-pointer metadata in a shadow memory space to provide memory layout compatibility. Conceptually, every word in memory has bounds and 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 4a), for every load instruction the Watchdog hardware injects

- a `check` µop to perform the check,
- a µop to perform the load of the actual value into the register, and
- a `shadow-load` µop to load the metadata from the shadow memory for every load.

Stores are handled analogously (Figure 4b). Watchdog requires that pointers are word aligned (as is required by some ISAs and is generally true with modern compilers for all ISAs), which allows the `shadow-load/store` µops to access the shadow space via an aligned wide 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 operating system's usual address translation and page allocation mechanisms. Current 64-bit x86 systems support 48-bit virtual addresses, so 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` and the `shadow-store` µop to convert an address to a shadow space address via simple bit selection and concatenation.

Implementation

Watchdog's checking and metadata propagation introduce performance overhead even when implemented in hardware. Therefore, Watchdog employs implementation optimizations to mitigate the performance overheads of such checking and streamline the implementation.

Reducing checking overhead

Watchdog performs a spatial (bounds) check and a temporal (use-after-free) check before every memory access by adding µops as shown in Figure 2b. This checking could

be implemented by injecting two μops, a bounds check μop and a temporal check μop, or by injecting a single μop that performs both checks in parallel. A bounds check operation is relatively inexpensive because it performs just two inequality comparisons and requires no additional memory accesses. In contrast, the use-after-free check performs an equality comparison and a memory access, which increases contention for limited cache ports.

To mitigate this impact, Watchdog 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 and 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 2c), has its own small translation look-aside buffer (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 (such as one that is 4 Kbytes) is effective because

- lock locations (8 bytes per object currently allocated) are small relative to the average object size, and
- the lock locations region has little fragmentation and exhibits reasonable spatial locality because lock locations are reallocated using a last-in, first-out (LIFO) free list.

Cache misses are handled just like misses in the data cache.

Reducing metadata accesses with pointer identification

Watchdog maintains bounds and identifier metadata with every pointer in a register or in memory. However, binaries for standard ISAs don't provide explicit information about which operations manipulate pointers. In the absence of such information, propagating metadata on every memory operation would require many extra memory operations to access the shadow space and would result in substantial performance degradation. However, employing one of two techniques for identifying pointer operations reduces the number of accesses to the metadata space.

Conservative pointer identification. To enable Watchdog to operate with reasonable overhead and 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 makes the conservative assumption that only a 64-bit load/store to an integer register may be a pointer operation, whereas floating-point load/stores and subword memory accesses are deemed nonpointer operations for which metadata manipulation μops are not inserted. This approach classifies 31 percent of memory operations as potentially loading/storing a pointer and is reasonably effective.[16]

ISA-assisted pointer identification. Although the conservative heuristic is effective and requires no ISA modifications, pointer operations can be identified more precisely by extending the ISA to allow the compiler to mark the load and store instructions that are manipulating pointers. The compiler, which generally knows which operations are manipulating pointers, is then responsible for selecting the proper load/store variant during code generation. For any statically ambiguous case, the compiler would conservatively select the memory operation variant that performs the metadata operations. This ISA-assisted pointer identification method reduces the number of memory accesses classified as pointer operations to 18 percent.

Decoupled register metadata

As described thus far, Watchdog copies metadata along with each arithmetic operation in registers. Although such register metadata copies can be implemented by maintaining widened registers, such a design suffers from two inefficiencies. First, every register write (and most register reads) accesses the sidecar metadata, which increases

the number of bits read and written to the register file. Although this isn't necessarily a performance problem, it could be an energy concern. Second, a more subtle issue is that treating register data and 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 `shadow-load` μop. These serializations have several detrimental effects, including increasing the load-to-use penalty of pointer loads, stalling subsequent instructions if either of the loads miss in the cache, and limiting the memory-level parallelism by serializing the load and the metadata load.

Decoupled register data and 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 64-bit register for data and one 256-bit register for the metadata. As a result of this change, individual μops generally operate on either the data or the metadata, which removes the serialization caused by partial register writes of monolithic registers. Once decoupled, the metadata propagation and checking are almost entirely removed from the critical path of the program's dataflow graph.

Metadata propagation for decoupled metadata. 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 (for example, the output of a subword operation or a divide is not a valid pointer), and 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 that selects the metadata from whichever register has valid metadata. If the ISA was further extended to allow the compiler to annotate such instructions, these `select` μops could also be eliminated.

Modified register renaming. Watchdog performs metadata propagation by modifying the register renaming hardware to reduce the number of extra μops inserted. Watchdog actually inserts μops in only one of the three cases we described earlier. In the other cases (copying the metadata or setting it to invalid), Watchdog uses previously proposed modifications to register renaming logic to handle these operations completely within the pipeline's register rename stage. Watchdog extends the map table 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 map table 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.[15] To ensure that this physical register is not freed until all the mappings in the map table are overwritten, these physical registers are reference counted. Figure 5 illustrates the decoupled register metadata and operation of Watchdog with extensions to the map table.

Summary of hardware changes

Figure 6 illustrates the changes made to the traditional out-of-order processor core to perform pointer-based checking with disjoint metadata. The number of additional structures that we need to add to an already existing out-of-order core to attain Watchdog's functionality is small.

Watchdog evaluation

The experimental evaluation of Watchdog highlights its effectiveness in preventing security exploits and its low performance overheads to provide comprehensive hardware-enforced memory safety.

Efficacy in preventing security vulnerabilities

To evaluate Watchdog's effectiveness in preventing memory safety-based security exploits, we ran test cases with heap-related buffer overflows (common weakness enumeration or CWE-122) and use-after-free

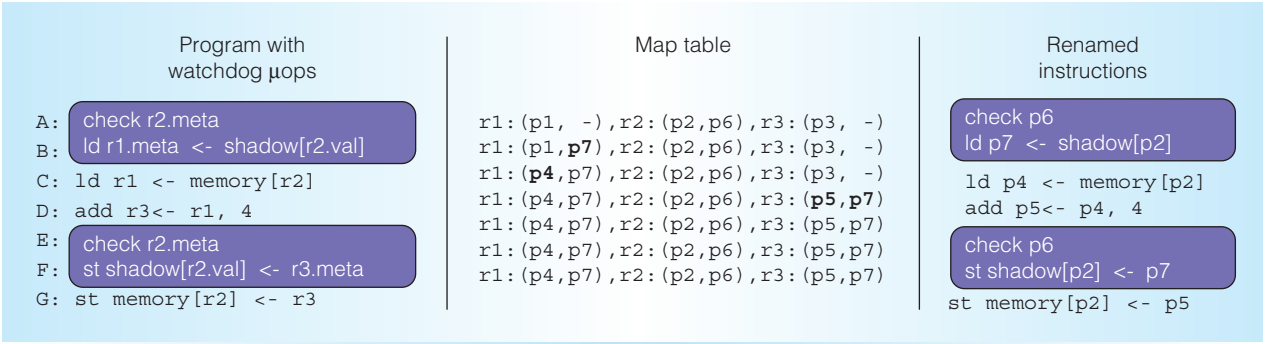

Figure 5. Register renaming with Watchdog µops and extensions to the map table. Watchdog inserted µops are shaded. The map table is represented by a tuple for each register. *r:(a,b)* means logical register *r* maps to physical register *a* according to the regular map table mapping and the logical register *r* maps to a 256-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).

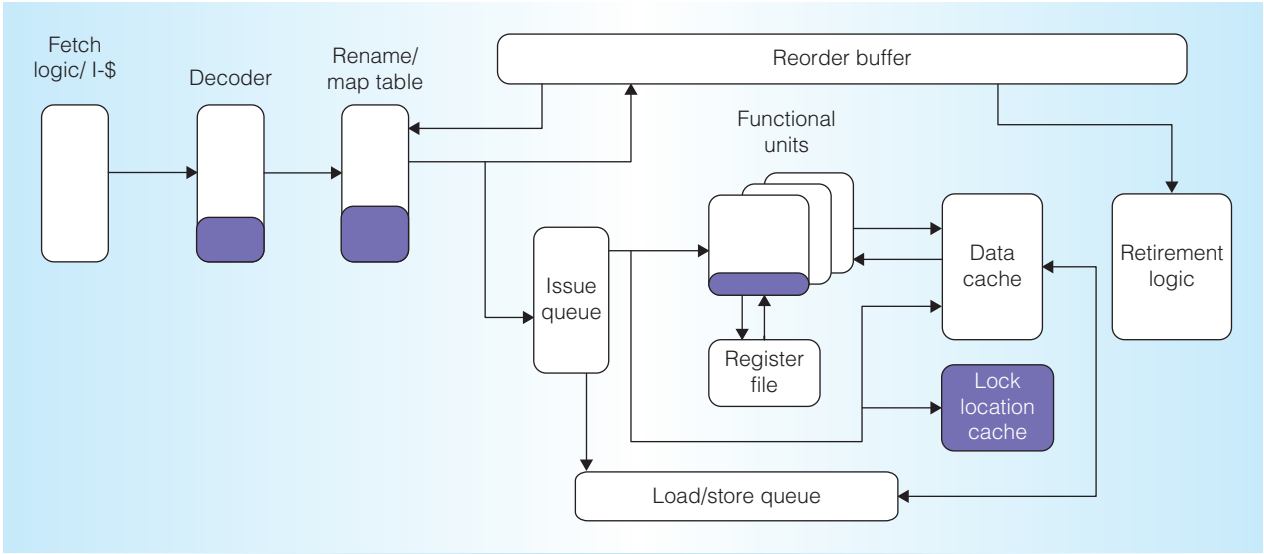

Figure 6. Changes made to the processor core with Watchdog in comparison to a traditional out-of-order core. Watchdog-specific modifications are shaded.

vulnerabilities (CWE-416 and CWE-562) from the US National Institute of Standards and Technology (NIST) Juliet Test Suite for C/C++ (http://samate.nist.gov/SRD), which are modeled after various memory safety errors reported in the wild. Watchdog successfully detected the memory safety errors in all test cases without any false positives.

Performance evaluation

To evaluate the benefits of hardware extensions, we used an x86-64 simulator, which executes the user-level portions of statically linked 64-bit x86 programs. We used applications from the System Performance Evaluation Corporation (SPEC) benchmark suite. Figure 7 presents the percentage of execution time overhead of Watchdog over a baseline without any Watchdog instrumentation. The graph contains a pair of bars for each benchmark. The height of the left and right bars represents the overhead of Watchdog with one check µop and two check µops, respectively. On average, Watchdog enforces

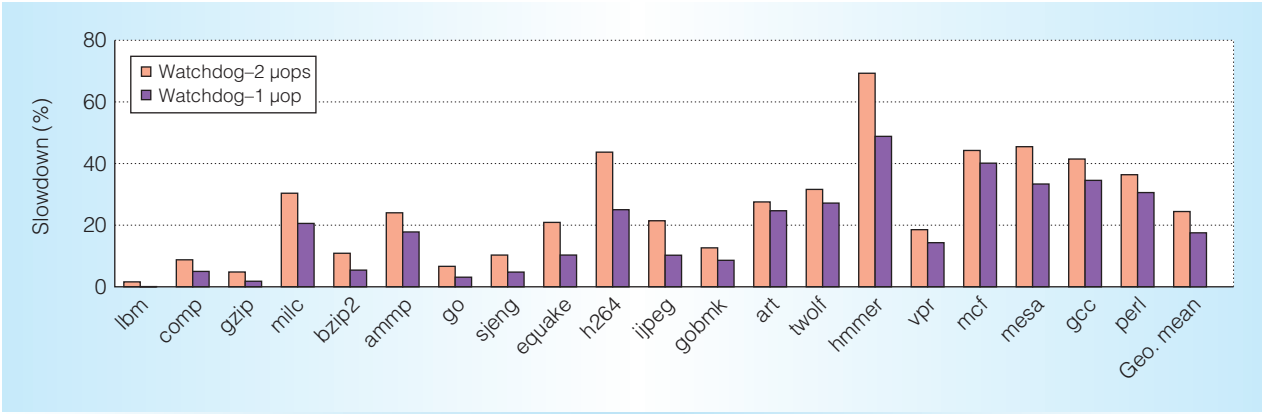

Figure 7. Watchdog's overhead to enforce comprehensive memory safety (detect both buffer overflows and use-after-free errors) with a single check µop and two check µops with ISA-assisted pointer identification. Shorter bars are preferable because they represent lower runtime overheads.

comprehensive memory safety with performance overheads of 18 percent and 24 percent with one check µop and two check µops, respectively. A detailed analysis of the performance overheads and the sensitivity studies are available elsewhere.[16] These overheads are an order of magnitude better than prior binary instrumentation and binary translation schemes, lower than the comprehensive software approaches (which often have 2× overhead), and are likely low enough to use in many deployment scenarios.[16]

Current mechanisms for preventing low-level security vulnerabilities consist largely of a patchwork of mitigation techniques that address the symptoms of vulnerabilities. Rather than treating the symptoms, Watchdog directly addresses the root cause—the lack of memory safety. The Watchdog hardware approach demonstrates that it is possible to enforce comprehensive spatial and temporal safety with low performance overheads by using pointer-based checking with disjoint metadata on legacy C programs. Watchdog's efficient and streamlined implementation is crucial to obtain low performance overheads.

In this work, the hardware is largely responsible for enforcing memory safety. An alternative approach is to more evenly divide the responsibilities between the hardware and the compiler by directly exposing Watchdog's µops as new instructions and shifting the responsibility of inserting checking operations to the compiler. Our experience in building pointer-based checking in various parts of the tool chain suggests that such a hardware/software co-design approach can provide low overhead checking while further reducing the invasiveness of the hardware modifications. MICRO

Acknowledgments

This research was funded in part by the US Government. 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 US Government. This research was funded in part by DARPA contract HR0011-10-9-0008, ONR award N000141110596, NSF grants CNS-1116682, CCF-1065166, and CCF-0810947, and donations from Intel Corporation.

References

1. J. Pincus and B. Baker, ''Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns,'' *IEEE Security & Privacy,* 2004, vol. 2, no. 4, pp. 20-27.
2. P. Porras, H. Saidi, and V. Yegneswaran, *An Analysis of Conficker's Logic and Rendezvous Points,* tech. report, SRI Int'l, Feb. 2009.
3. J. Devietti et al, ''Hardbound: Architectural Support for Spatial Safety of the C Programming Language,'' *Proc. 13th Int'l Conf.*

Architectural Support for Programming Languages and Operating Systems, ACM, 2008, pp. 103-114.
4. S. Nagarakatte, M.M.K. Martin, and S. Zdancewic, ''Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety,'' *Proc. 39th Ann. Int'l Symp. Computer Architecture,* ACM, 2012, pp. 189-200.
5. S. Nagarakatte et al, ''SoftBound: Highly Compatible and Complete Spatial Memory Safety for C,'' *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* ACM, 2009, pp. 245-258.
6. S. Nagarakatte et al, ''CETS: Compiler Enforced Temporal Safety for C,'' *Proc. Int'l Symp. Memory Management,* ACM, 2010, pp. 31-40.
7. T.M. Austin, S.E. Breach, and G.S. Sohi, ''Efficient Detection of All Pointer and Array Access Errors,'' *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* ACM, 1994, pp. 290-301.
8. W. Chuang, S. Narayanasamy, and B. Calder, ''Accelerating Meta Data Checks for Software Correctness and Security,'' *J. Instruction-Level Parallelism,* June 2007, pp. 1-26.
9. T. Jim et al, ''Cyclone: A Safe Dialect of C,'' *Proc. USENIX Ann. Technical Conf.,* USENIX Assoc., 2002, pp. 275-288.
10. G.C. Necula et al, ''CCured: Type-Safe Retrofitting of Legacy Software,'' *ACM Trans. Programming Languages and Systems,* May 2005, pp. 477-526.
11. H. Patil and C.N. Fischer, ''Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs,'' *Software—Practice & Experience,* vol. 27, no. 1, 1997, pp. 87-110.
12. W. Xu, D.C. DuVarney, and R. Sekar, ''An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs,'' *Proc. 12th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* ACM, 2004, pp. 117-126.
13. K. Ganesh, ''Pointer Checker: Easily Catch Out-of-Bounds Memory Accesses,'' Intel, 2012, http://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf.
14. M.L. Corliss, E.C. Lewis, and A. Roth, ''DISE: A Programmable Macro Engine for Customizing Applications,'' *Proc. 30th Ann. Int'l Symp. Computer Architecture,* ACM, 2003, pp. 362-373.
15. V. Petric, T. Sha, and A. Roth, ''RENO: A Rename-Based Instruction Optimizer,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture,* IEEE, 2005, pp. 216-225.
16. S. Nagarakatte, ''Practical Low-Overhead Enforcement of Memory Safety for C Programs,'' doctoral thesis, Computer and Information Sciences Dept., Univ. of Pennsylvania, 2012.

Santosh Nagarakatte is an assistant professor in the Department of Computer Science at Rutgers University. His research interests include hardware-software interfaces spanning compilers, programming languages, and computer architecture. Nagarakatte has a PhD in computer science from the University of Pennsylvania.

Milo M. K. Martin is an associate professor in the Department of Computer and Information Science at the University of Pennsylvania. He is also the faculty coleader of Penn's Computer Architecture and Compilers Group. His research interests include scalable microarchitectures, multiprocessor computer architectures, memory systems, and verification. Martin has a PhD in computer science from the University of Wisconsin-Madison.

Steve Zdancewic is an associate professor in the Department of Computer and Information Science at the University of Pennsylvania. His research interests include programming languages, type systems, theorem proving for verified compilers, and linear logics. Zdancewic has a PhD in computer science from Cornell University.

Direct questions and comments about this article to Santosh Nagarakatte, Department of Computer Science, Rutgers University, SAS-Computer Science, 110 Frelinghuysen Road, Piscataway, NJ, 08854; santosh.nagarakatte@cs.rutgers.edu.