ARMOR: A Recompilation and Instrumentation-Free Monitoring Architecture for Detecting Memory Exploits

Alex Grieve, Michael Davies, Phillip H. Jones, *Member, IEEE*, and Joseph Zambreno, *Senior Member, IEEE*

Abstract—Software written in programming languages that permit manual memory management, such as C and C++, are often littered with exploitable memory errors. These memory bugs enable attackers to leak sensitive information, hijack program control flow, or otherwise compromise the system and are a critical concern for computer security. Many runtime monitoring and protection approaches have been proposed to detect memory errors in C and C++ applications, however, they require source code recompilation or binary instrumentation, creating compatibility challenges for applications using proprietary or closed source code, libraries, or plug-ins. This paper introduces a new approach for detecting heap memory errors that does not require applications to be recompiled or instrumented. We show how to leverage the calling convention of a processor to track all dynamic memory allocations made by an application during runtime. We also present a transparent tracking and caching architecture to efficiently verify program heap memory accesses. Performance simulations of our architecture using SPEC benchmarks and real-world application workloads show our architecture achieves hit rates over 95 percent for a 256-entry cache, resulting in only 2.9 percent runtime overhead. Security analysis using a software prototype shows our architecture detects 98 percent of heap memory errors from selected test cases in the Juliet Test Suite and real-world exploits.

Index Terms—Memory allocation tracking, hardware architecture, range cache, vulnerability testing

1 Introduction

Software memory errors, such as buffer overflows and use-after-free errors, are critical threats to computer system security. Applications built using low level languages that allow arbitrary pointer arithmetic, casting, and manual memory management, such as C and C++, are particularly susceptible to memory errors. Exploiting these errors enables attackers to read or write arbitrary memory locations, alter control flow of a target application, or even take complete control of a system. Numerous hardware and software-based approaches to detect and prevent exploitation of memory errors have been proposed, and several have been integrated into modern systems [1], [2], [3]. However, current protection mechanisms can still be circumvented as shown by recent real-world exploits [4], [5], [6], [7].

Many runtime monitoring and protection approaches have been proposed to detect memory errors in C and C++ software applications. Previous software-based techniques monitor memory accesses by inserting checks into program source code [8], augmenting compilers to insert checks at compile time [9], [10], or instrumenting the application binary to perform memory access checking at runtime [11], [12]. Several new, memory safe programming languages based on C and C++ have also been proposed [13]. Software-based techniques can detect many classes of memory errors and exploits, however, they require recompilation of source code and impose large performance and memory overheads that often outweigh the security protection afforded by the technique.

Previous hardware-based approaches address the performance limitations of software-based techniques with custom memory checking architectures [14], [15], [16], [17], [18], [19], [20], [21]. Although custom architectures significantly reduce runtime overheads, they still require all source code to be recompiled to make use of the special hardware. The source code recompilation requirement, common to both hardware and software approaches, is not practical with legacy software or software that utilizes proprietary, third-party code, libraries, plug-ins, or applications. These performance and compatibility limitations have prevented the widespread adoption of many proposed solutions.

This paper introduces a novel hardware architecture for tracking dynamic memory allocations and securing heap memory accesses that does not require recompilation of source code or instrumentation of the target application. We show how to leverage the calling convention of a processor to track all dynamic memory allocations made by an application during runtime. We detect memory errors by proposing a caching architecture that verifies all heap memory

• *The authors are with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011. E-mail: {argrieve, mdavies, phjones, zambreno}@iastate.edu.*

Manuscript received 31 May 2017; revised 12 Nov. 2017; accepted 14 Feb. 2018. Date of publication 19 Feb. 2018; date of current version 7 July 2018. (Corresponding author: Joseph Zambreno.)
Recommended for acceptance by G. Bertoni.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2807818

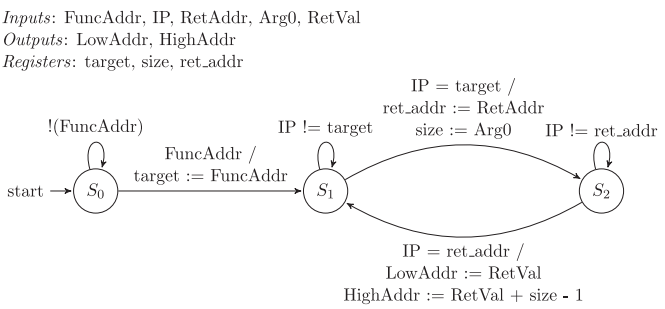

Fig. 1. High level state machine depicting our dynamic memory allocation tracking approach.

accesses are to valid memory addresses allocated to the program. We evaluate our architecture by creating a simulation prototype using the Intel Pin framework. Performance tests using SPEC 2006 benchmarks and real-world applications show cache hit rates over 95 percent are achievable for a 256-entry cache, resulting in just 2.9 percent runtime overhead. Security experiments using test cases from the NSA Juliet Test Suite and real-world exploits show our architecture successfully detects 98 percent of heap memory errors spanning seven different classes.

The remainder of this paper is organized as follows. Section 2 introduces our heap memory protection approach and analyzes its security effectiveness using benchmarks and real-world exploits. Section 3 describes the hardware architecture that implements our heap memory protection approach, and its performance is analyzed using SPEC benchmarks and real-world application workloads. Section 4 discusses implementation considerations for integrating our hardware into existing systems. Section 5 reviews related work in the area of C and C++ software protection approaches, and Section 6 offers our conclusion and outlines future work.

2 Memory Protection Approach

In this section, we present problem constraints and assumptions for our memory protection approach. We describe how to leverage a processor's calling convention to track dynamic memory allocations made using library functions and system calls, and explain our technique for verifying memory accesses at runtime. We also evaluate the security effectiveness of our approach using benchmarks and real-world exploits.

2.1 Constraints and Assumptions

The primary goal of this work is to develop a memory safety solution that is fully compatible with existing software applications, libraries, and plug-ins to facilitate widespread adoption. To achieve this compatibility goal, potential solutions will not require source code to be recompiled and application binaries will not be instrumented or otherwise modified. Binaries, libraries, and plug-ins may be parsed and examined, but we assume any type of binary or source code changes violate the compatibility constraint.

Our heap memory allocation monitoring approach depends on two requirements essential for its operation. First, the heap memory allocation monitor needs read-only access to the processor's instruction stream and architectural register file. While specific registers within the register file will vary depending on the processor's instruction set architecture and calling convention, read access to registers containing function arguments, return values, the stack pointer, and the instruction pointer are required for heap memory allocation tracking. Access to the processor's instruction stream is also required to capture system call instructions and memory access instructions.

The second requirement is the virtual memory addresses of dynamic memory allocation functions, such as `malloc` and `free`, are known at runtime. Applications that dynamically link libraries containing dynamic memory allocation functions at runtime will have these addresses resolved by the dynamic linker. Locating function addresses in statically compiled applications that have not been stripped of debugging information, such as function labels or symbols, can be achieved by inspecting the binary. Recent fingerprinting and pattern matching techniques successfully locate library functions, including `malloc` and `free`, in statically compiled binaries with debugging information removed [22].

To simplify the description of our heap memory allocation monitor as well as our prototype and experiments, we make two assumptions about the instruction set architecture and its calling convention. First, we assume that the instruction set architecture uses a hardware stack for subroutine information storage. The hardware stack is contiguous in memory, has a fixed base address, and an architectural stack pointer register holds the address of the current end of the stack. Second, we assume all arguments to dynamic memory allocation functions are passed in architectural registers, including the return address for the dynamic memory allocation function call.

2.2 Dynamic Memory Allocation Tracking

The high level state machine in Fig. 1 illustrates our heap memory tracking approach for a dynamic memory allocation function that takes a single size argument, such as `malloc`. Our tracking monitor begins in state S_0 and waits to receive the virtual address of the memory allocation function via the *FuncAddr* input. If the target application is dynamically linked, then the dynamic loader supplies the allocation function address after its location is initially resolved. If the tracked program is statically linked, the kernel program loader supplies the allocation function address by parsing debug information in the binary or applying fingerprinting techniques described in [22]. When the monitor receives the allocation function's virtual address, the address is stored in the monitor's *target* register, and the monitor moves into state S_1.

In state S_1, the monitor waits for the target application to make a dynamic memory allocation by continuously comparing the instruction pointer with the virtual address of the allocation function stored in the monitor's *target* register. When the application calls the allocation function, the monitor exploits the calling convention of the processor to determine the amount of memory requested by the target program. For example, the size argument to the allocation function is placed in the R0 register on 64-bit ARM (AArch64) processors, the A0 register on 64-bit RISC-V processors, and the RDI register for 64-bit x86 processors using the System V AMD64 ABI. The monitor reads the size

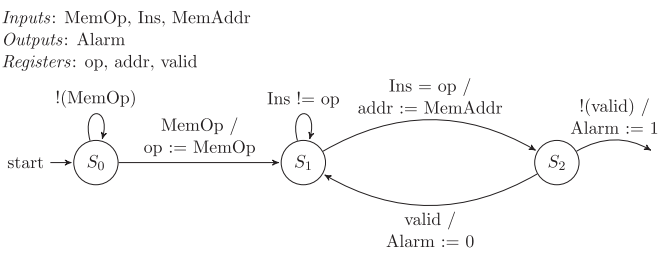

Fig. 2. High level state machine illustrating our memory checking approach.

argument from the appropriate architectural register and stores the value into its *size* register. The monitor also reads the procedure return address from the correct architectural register, such as the LR register on 64-bit ARM (AArch64) processors or the RA register on 64-bit RISC-V processors, and saves the address into its *ret_addr* register. Then, the monitor moves into state S_2.

In state S_2, the monitor waits for the allocation function call to return by comparing the instruction pointer with the return address saved in the *ret_addr* register. When the call to the allocation function returns, the monitor leverages the calling convention of the processor to extract the virtual address returned by the allocation function. For example, the return value is placed in the R0 register on 64-bit ARM (AArch64) processors, the A0 register on 64-bit RISC-V processors, and the EAX register for 64-bit x86 processors using the System V AMD64 ABI. Finally, the monitor calculates the virtual address range allocated to the application using the virtual address in the return value register and the previously saved size argument, outputs the lower and upper addresses of the range, and returns to state S_1.

Access to the processor's architectural register file allows the monitor to track other memory allocation functions with multiple size parameters or size parameters that appear in different locations in the function parameter list. For example, the `calloc` memory allocation function requires the number of elements and individual element size as function parameters. Instead of reading a single argument register, such as *Arg0* shown in Fig. 1, the monitor computes the size of the memory allocation by reading the first two argument registers and multiplying their values. Other memory allocation functions place the size parameter in different positions in the parameter list. For example, the second parameter to the `mmap` function is the allocation size parameter. Because the monitor has read access to all processor registers, it can be configured to read the appropriate register to obtain the size of the allocation requested by the program.

Although it is not explicitly shown in Fig. 1, before the monitor outputs a new range of allocated virtual addresses, it validates the return value of the memory allocation or deallocation function to verify the memory allocation was successful. If an error value, such as a null pointer, is returned by an allocation function, the monitor does not output a virtual address range. This step is particularly important for memory reallocation functions, such as `realloc`, that can allocate, deallocate, and move virtual memory regions depending on the arguments passed to the function. By checking the arguments and return values, the monitor determines the exact changes made to the application's heap memory region.

The monitoring approach can be slightly modified to support direct system calls. Modern applications tend to issue system calls using wrapper functions instead of invoking the kernel directly. However, it is common for the program loader and dynamic linker to issue many direct system calls during program start up and when mapping dynamically shared libraries into a program's virtual address space. To track memory allocations made using direct system calls, the monitor checks the processor's instruction stream for a system call instruction. The *FuncAddr* input in Fig. 1 is set to the opcode of the system call instruction and is checked against the opcode of the current instruction instead of the instruction pointer. The monitor is configured to read the register containing the system call number along with the registers containing allocation size arguments, and the system call number is checked during the monitor's verification stage.

Our heap memory tracking approach has several advantages. Its flexible design enables monitoring of a diverse set of dynamic memory allocation functions and system calls with varied parameter lists. By tracking at the entry and exit point of the initial memory allocation function call, our monitoring approach is unaffected by implementation characteristics of allocation functions such as memory pooling, system calls, or recursion. Monitoring calls to common dynamic memory allocation functions also enables tracking of memory regions allocated by dynamically shared libraries, plug-ins, or other dependencies loaded at runtime. Most importantly, our heap memory tracking approach transparently tracks the heap memory addresses allocated to an application without source code modification, recompilation, or binary instrumentation.

2.3 Validating Memory Accesses

Maintaining a list of all heap memory addresses allocated to an application alone does not provide any additional security against memory errors and exploits. To enforce memory safety, we verify that all heap memory accesses are within the bounds of a valid heap memory region currently allocated to the application. A heap memory safety violation occurs when a program reads or writes to a memory address outside the bounds of all valid heap memory allocations and will trigger an exception.

Our heap memory checking approach is illustrated by the high level state machine in Fig. 2. In state S_0, the monitor waits for the program loader to supply an opcode for a memory access instruction, such as a load or store instruction. The monitor stores this opcode into its private *op* register and proceeds to state S_1. In state S_1, the processor's instruction stream is inspected. When the opcode of the processor's current instruction matches the value stored in the monitor's *op* register, the monitor extracts the virtual memory address from the appropriate register (designated by the instruction) and stores the address into its *addr* register. In state S_2, the memory address stored in the *addr* register is checked against all valid dynamically allocated memory regions recorded using our tracking approach. If the memory address is within the bounds of a valid heap memory region, the monitor returns to state S_1. Otherwise, a heap memory safety violation is detected, and an alarm is raised.

The memory checking approach depicted in Fig. 2 verifies memory accesses made by an application at runtime

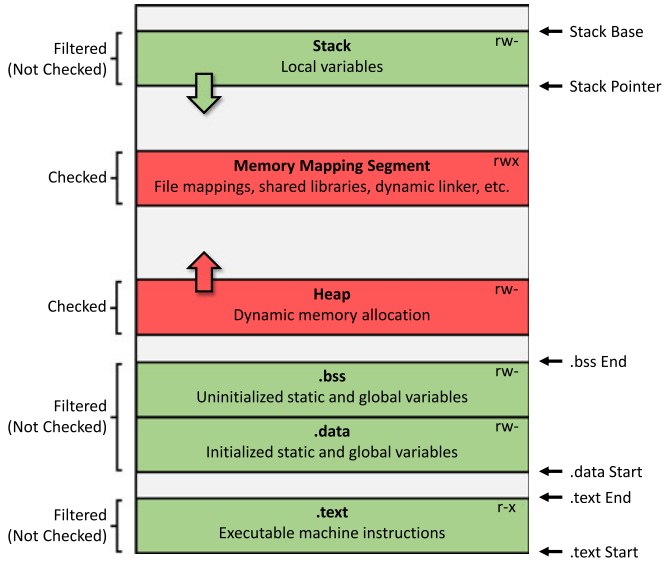

Fig. 3. Memory accesses to filtered regions are not checked for memory safety violations.

without source code modification, recompilation, or binary instrumentation. However, checking memory accesses to addresses that are not dynamically allocated will result in false positive memory safety violations. To prevent exceptions from being raised by false positives, addresses located in statically allocated memory regions should not be checked. The monitor is made aware of these regions using a set of filtering registers and by reading the processor's architectural registers. Memory addresses contained within the filtered regions shown in Fig. 3 are excluded from heap memory access checking.

As shown in Fig. 3, the program's stack is a memory region that is filtered. At startup, the program loader allocates the application's stack and supplies the base address of the stack to the monitor, and the monitor stores the address in one of its filtering registers. To determine the current end address of the stack, the monitor reads the value in the processor's stack pointer register. Using its filter register containing the stack base address and the processor's stack pointer register, the monitor filters memory accesses to addresses residing on the program's stack.

Memory accesses to the program's text and data sections are also filtered. Modern computing systems support Data Execution Prevention or Write-XOR-Execute policies that halt program execution when memory in the text section is written. Although reads to the text section are still permitted, the same information can be obtained by disassembling the application binary. Static and global variables are stored in the data section and should be accessible by the application without generating false positive heap memory safety violations. Segments that support dynamic linking, such as the Global Offset Table and the Procedure Lookup Table, are also located in the program's data section and are accessed by the program and the dynamic linker at runtime. The kernel program loader supplies the monitor with the upper and lower addresses of the text and data sections for statically compiled applications, and the dynamic linker supplies the monitor with the location of the text and data sections for dynamically compiled programs. The monitor places the upper and lower address of the text and data sections into four filter registers and excludes memory accesses to the text and data sections from heap memory checks.

In addition to excluding particular memory regions, we also suspend memory checking during two points of program execution. First, we do not check memory addresses before entry into the program's main function or after exiting from the program's main function. This allows the program loader or dynamic linker to perform setup or teardown routines without raising false positives. It is common for these routines to access memory regions beyond the end of the stack to communicate with the operating system kernel. If checked, these memory accesses raise exceptions related to initialization and cleanup procedures instead of memory safety errors caused by the target application.

Memory checks are also suspended during execution of dynamic memory allocation functions. This allows allocation functions to manage internal information located in regions not explicitly allocated to the target application without raising any memory safety exceptions. Our tracking monitor can be easily extended to set a flag while it waits for an allocation function to finish.

TABLE 1
Security Results Using the Juliet Test Suite

CWE	Description	Detected	# Tests
122	Heap-based Buffer Overflow	60	62
124	Buffer Underwrite	10	10
126	Buffer Overread	6	6
127	Buffer Underread	10	10
415	Double Free	17	17
416	Use-after-free	18	18
590	Free Memory Not on Heap	57	57

2.4 Security Analysis

To test our dynamic memory allocation tracking and memory checking monitor, we developed a software prototype using the Pin instrumentation framework [23] for 64-bit Linux systems. We used Pin to locate the `malloc`, `calloc`, `realloc`, `free`, `mmap`, `mremap`, and `munmap` functions in the target application and replace the signature of each procedure with its own custom wrapper function. System calls that dynamically allocate memory, specifically `mmap`, `mremap`, `munmap`, and `brk`, are also tracked using similar system call wrapper functions. This instrumentation has the effect of invoking the wrapper function when the application calls a dynamic memory allocation routine, allowing our prototype to capture size arguments, invoke the wrapped routine, and record the address ranges of dynamically allocated regions. To enforce memory safety, we used the Pin framework to verify all committed memory instructions that access addresses outside of filtered regions are within the bounds of a valid heap memory region allocated to the program.

We analyzed the security protection of our monitoring approach by testing the Pin prototype against test cases from the National Security Agency's (NSA) Juliet Test Suite for C/C++ [24]. The results listed in Table 1 show our monitoring approach detects all out of bounds memory accesses to dynamically allocated buffers including buffer underwrites, overreads, and underreads. Our prototype also detects

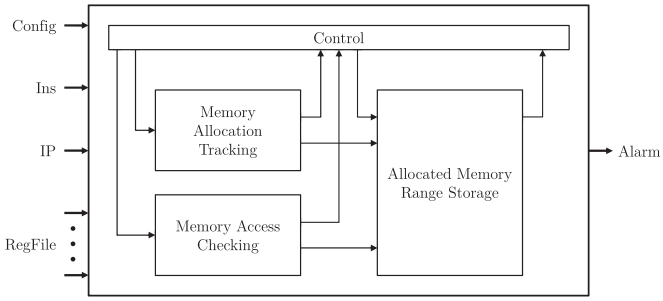

Fig. 4. Top level design of our hardware architecture.

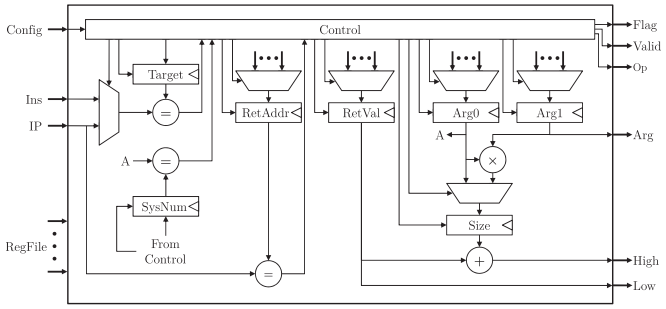

Fig. 5. Hardware architecture of a dynamic memory allocation tracking submodule.

several types of invalid memory deallocations, such as freeing memory not allocated on the heap and double freeing an allocated memory region. The results also demonstrate our approach catches temporal memory violations such as use-after-free errors. In addition, our prototype did not generate any false positive memory violations during testing.

In addition to the Juliet Test Suite, we also examined the effectiveness of our memory monitoring approach in detecting real-world attacks that exploit memory errors. Using our Pin prototype, we monitored applications containing the Heartbleed and GHOST vulnerabilities. These two vulnerabilities are significant because they are remotely exploitable, reliable, and affected a large number of systems for several years before being publicly disclosed [25], [26]. Exploiting the Heartbleed and GHOST vulnerabilities requires reading and writing memory beyond the bounds of a dynamically allocated buffer, respectively. Our software prototype detected exploitation of both vulnerabilities, suggesting that our memory protection approach can detect attacks against real-world systems that exploit unknown or undisclosed vulnerabilities in production software.

Although our memory protection approach detected over 98 percent of memory errors during security testing, our technique shares the same limitations as previously proposed memory safety approaches. Our technique cannot identify sub-object boundaries within a dynamically allocated region of memory and therefore cannot detect memory errors that occur inside the tracked memory region. This limitation is common to all protection approaches that track allocations at object-bounds granularity and is the reason for the two undetected memory errors in Table 1. Similar to previous temporal memory protection solutions, memory accesses made by dereferencing dangling pointers are not detected if the memory region to which they point is reused. Finally, memory errors and attacks targeting filtered regions, such as the stack or data section, will not be detected by our monitoring approach.

3 Hardware Architecture

This section introduces our hardware architecture that implements our memory protection approach. We describe a hardware design that tracks dynamic memory allocations and checks memory accesses to allocated memory regions. Profiling experiments used to design our architecture for low memory and performance overheads are also presented.

3.1 Top Level Design

The top level design of our monitoring hardware implementation is illustrated in Fig. 4. The *Config* input accepts configuration options for dynamic memory allocation tracking such as virtual addresses of allocation functions to monitor, system call numbers, processor registers containing relevant arguments, and allocation size computation. The *Config* port also accepts commands that set up memory access checking such as opcodes of memory instructions, processor registers containing effective memory addresses, and values to place in filtering registers. Configuration of the monitor should only be allowed when the processor is operating in a privileged state, such as kernel mode.

Other ports on our hardware monitoring implementation include the *Ins*, *IP*, and *RegFile* inputs and the *Alarm* output. The *Ins* input corresponds to the instruction currently being executed by the processor and is used for tracking system calls that dynamically allocate memory and checking instructions that access memory. *IP* is the current value of the instruction pointer register and is used by the memory allocation tracking hardware to detect and record dynamic memory allocations. To simplify our hardware design, we require only committing instructions and instruction pointer values to be placed on the *Ins* and *IP* inputs, respectively. This prevents incorrect tracking of dynamically allocated memory regions during speculative execution, and also prevents false positive memory violations from being raised during speculative memory accesses. The *RegFile* input corresponds to the processor's integer register file and is necessary for capturing allocation function arguments, system call numbers, and memory addresses. Finally, the *Alarm* output is set when a memory access violation occurs.

The *Memory Allocation Tracking* component shown in Fig. 4 contains several hardware submodules, each configured for detecting dynamic memory allocations made by a single function routine or system call. The *Memory Access Checking* component also contains multiple submodules that individually check the opcode of a single memory access instruction. The *Control* unit shown in Fig. 4 is used to configure the memory allocation tracking and memory access checking hardware. The *Control* unit also regulates access to the *Allocated Memory Range Storage* component containing the virtual addresses of dynamically allocated memory regions, and sets the *Alarm* output when a memory access violation occurs. The design of the memory tracking, checking, and storage components is discussed in the following sections.

3.2 Memory Allocation Tracking Architecture

The top level memory allocation tracking component contains several of the hardware submodules depicted in Fig. 5.

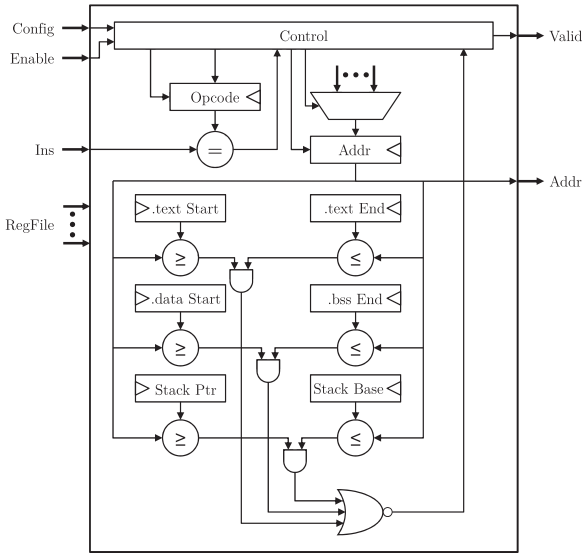

Fig. 6. Hardware architecture of a submodule that outputs memory addresses for checking.

Fig. 5 contains the necessary hardware elements to support tracking all of the allocation functions and system calls used in our software prototype. The *Config* input is used to configure all five multiplexers and the value contained in the *Target* register. Allocation functions can be tracked by placing the virtual address of the function in the *Target* register and configuring the vertical multiplexer to output the *IP* input. System calls can be monitored by placing the system call opcode in the *Target* register, writing the system call number of the system call to monitor in the *SysNum* register, and configuring the vertical multiplexer to output the *Ins* input.

The four horizontal multiplexers located at the top of Fig. 5 are configured to capture the correct return address, return value, and argument values from the processor's register file. The configuration of these multiplexers is ultimately determined by the processor's calling convention. The multiplexer located above the *Size* register selects the proper computation needed to determine the size of the dynamic memory allocation. Functions that have a single size argument, such as `malloc`, are configured to propagate the value in *Arg0* to the *Size* register. The other input to the multiplexer is selected when tracking allocations made using the `calloc` function where the size is determined by multiplying the arguments specifying the number of elements and the individual element size.

In addition to configuring the module to properly track a specific allocation function, the *Control* unit in Fig. 5 enables registers at the appropriate times to capture return addresses, return values, and function arguments. It also sets the *Flag* output high while an allocation routine is executing to suspend memory checking, and when the allocation function completes, the *Control* unit sets the *Valid* output high. Using the *Op* output, the *Control* unit reports the type of memory operation performed (allocation, deallocation, or reallocation) to facilitate proper modifications in the allocated memory storage module. The *High* and *Low* outputs contain the upper and lower virtual addresses of the newly allocated region. The *Arg* output is necessary for capturing arguments to deallocation routines, specifically the base address of a memory region to be deallocated. In the event that a reallocation routine results in the relocation of a memory region, *Arg* outputs the base address of the memory region prior to the relocation, allowing the storage module to properly update its contents.

3.3 Memory Checking Architecture

Fig. 6 illustrates the hardware design of a checking submodule contained within the top level memory access checking module. The opcode of a memory access instruction is supplied on the *Config* port, and the *Control* unit writes the value into the *Opcode* register. The location of the source register operand within the instruction is also specified using the *Config* port. The address ranges of filtered memory regions are externally supplied using the *Config* input, and the *Control* unit stores the upper and lower address of each filtered region in a set of filter registers.

When enabled, the memory checking submodule monitors the instruction stream for an instruction whose opcode matches the value in the *Opcode* register. When a match occurs, the *Control* unit reads the processor register specified by the instruction's source operand and stores the effective memory address in the *Addr* register. The memory address is then checked against the ranges stored in the filter registers. If the memory address is not in a filtered region, the *Valid* output is set, indicating the value on the *Addr* output port should be checked for a memory access violation.

It is important to note that for simplicity, Fig. 6 only shows the hardware necessary for supporting register indirect addressing. Because modern processors support many different addressing modes, the logic for determining the effective address can become quite complex. It may be more efficient to implement a mechanism by which the processor sends the effective address to the memory checking hardware directly.

3.4 Allocated Memory Storage Architecture

The following sections describes the hardware architecture for storing and retrieving addresses of dynamically allocated memory regions. The process of selecting a storage format with low memory overheads is presented, followed by an explanation of our hardware architecture. Performance experiments are carried out using different configurations of our architecture, and the results are discussed.

3.4.1 Storage Format

Before designing a hardware architecture for storage and retrieval of metadata describing dynamically allocated memory regions, we experimentally selected a storage format for the allocation metadata that minimizes memory overheads. The first format we considered stores a single bit of metadata for every dynamically allocated byte of virtual memory. This metadata bit approach has been used in previous work to store different types of metadata [11], [14], [15], [21], [27]. The second format we considered stores a lower and upper address pair that marks the start and end address of a dynamically allocated range of memory.

To evaluate the amount of memory required for both metadata bit and range pair storage formats, we created a profiling tool using Pin. Similar to our software prototype described in Section 2.4, our profiling tool wraps dynamic

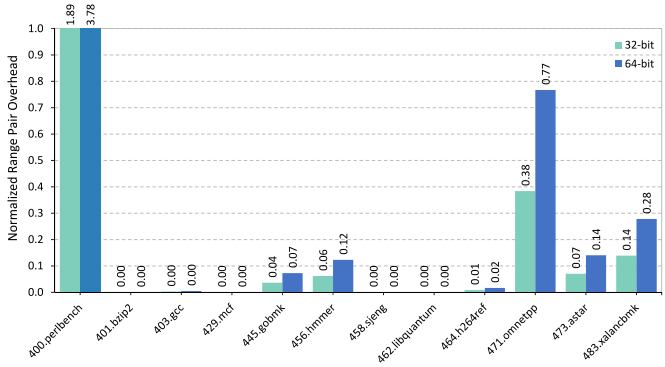

Fig. 7. Memory overhead of the range pair storage format normalized to metadata bit storage overhead for SPEC integer benchmarks.

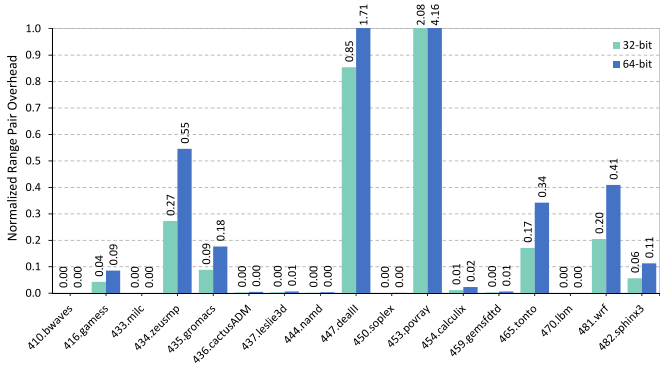

Fig. 8. Memory overhead of the range pair storage format normalized to metadata bit storage overhead for SPEC floating point benchmarks.

memory allocation functions and extracts the allocation size information. For each invoked dynamic memory allocation routine, the amount of overhead for storing metadata bits, 32-bit range pairs, and 64-bit range pairs is computed by the profiling tool and added to a running total. The average dynamic memory allocation size is also tracked to determine the storage format that requires less memory overhead on average.

Using our profiling tool, we ran the SPEC 2006 integer and floating point benchmarks and determined the total memory overheads of each storage format [28]. The memory overhead of storing metadata bits is a function of the size of the dynamic memory allocation, however, the overhead of storing a pair of addresses is constant for each memory allocation. For example, storing two 32-bit addresses requires 64 bits of memory overhead, and storing two 64-bit addresses requires 128 bits of overhead. Therefore, we expected the metadata bit format to require less storage overhead for benchmarks whose average allocation size is smaller than the constant amount required for storing a range pair.

Fig. 7 provides a visual representation of the 32-bit and 64-bit range pair overheads normalized to metadata bit overheads for each SPEC 2006 integer benchmark. The range pair format requires less storage overhead for benchmarks whose normalized value is less than 1.0, and the metadata bit format has a smaller memory footprint for benchmarks with a normalized value greater than 1.0. The 32-bit and 64-bit range pair format requires less storage overhead than the metadata bit format for all integer benchmarks except 400.perlbench. The average allocation size for 400.perlbench is close to the constant value required to store a range pair, indicating that the benchmark makes many small allocations that are more compactly represented using the metadata bit storage format. The remaining benchmarks have larger average allocation sizes and therefore have less overhead when the range pair format is used.

Fig. 8 shows the normalized memory overhead of the range pair storage format for each SPEC 2006 floating point benchmark. Normalized values below 1.0 indicate the range pair representation has a smaller memory footprint, and storing metadata bits requires less memory overhead for benchmarks with a normalized value above 1.0. The results for 453.povray are similar to 400.perlbench in that the benchmark has a small average allocation size and metadata bits are therefore a more efficient storage format. The results for 447.dealII show allocations are more compactly represented by 32-bit range pairs than metadata bits, but storing metadata bits has a smaller memory footprint than 64-bit range pairs. This outcome suggests 447.dealII makes a set of allocations that are more compactly represented as 32-bit range pairs but not 64-bit range pairs when compared with the metadata bit storage format. The remaining floating point benchmarks have a smaller memory footprint when storing allocated range information in both 32-bit and 64-bit range pair formats.

Overall, our profiling experiments demonstrate the range pair storage format requires less memory overhead than storing metadata bits for 91.3 percent of SPEC benchmarks. Based on these results, we elected to store dynamically allocated memory region information as a lower and upper address pair marking the start and end address of an allocated range of memory.

3.4.2 Range Cache Architecture

Before presenting our range cache architecture, we briefly discuss allocated range storage and lookup methods we decided not to pursue. Hashing approaches are not appropriate because a range lookup requires every memory address in a range pair to collide when hashed. Developing a hash function that produces collisions for arbitrarily sized address ranges within a large virtual address space is difficult. We also investigated a hardware implementation of a binary search tree. However, hardware search trees are not especially favorable because they require longer average lookup times than caches, and they typically do not exploit any program locality when the same range pair is accessed frequently. Previous work suggests that balancing and modifying trees in hardware requires a significant amount of control logic that increases the overall latency of the hardware tree [29]. We elected to pursue a caching architecture because caches exploit program locality that can improve overall lookup performance, and the control logic to support range lookup and modification is not overly complex.

Fig. 9 illustrates how a search operation is performed in our range cache architecture when checking a memory address for a violation. Each cache entry consists of two registers containing a lower and upper address of a range pair. The range cache is fully associative, and a cache hit occurs when the value of a checked address is between the lower and upper addresses of an entry in the cache. A range cache miss implies a memory access violation, however, not

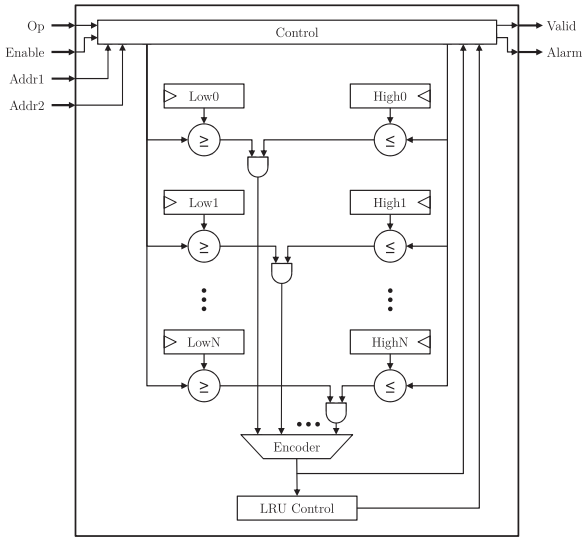

Fig. 9. Range cache hardware for search operations.

all allocated memory range pairs may be present in the range cache. If the cache misses, a second level range cache may be searched, or a software handler may be invoked to service the miss.

In addition to performing memory address searches, the range cache also supports update operations including range insertion, range deletion, and range modification. Fig. 10 shows the relevant hardware that implements range cache update functionality. The execution flow through the hardware elements is slightly different depending on the type of update operation. Range insertions write the new lower and new upper address into the least recently used (LRU) range cache entry in a single step. Range modifications are performed when an application invokes a dynamic memory reallocation or deallocation routine, and modifications are carried out in two steps. In the first step, the range cache is searched for the range entry to be modified, and the most recently used (MRU) index is updated with the entry that hit. In the second step, the MRU index is used to write the new low and high values to the range cache entry that hit during the first step.

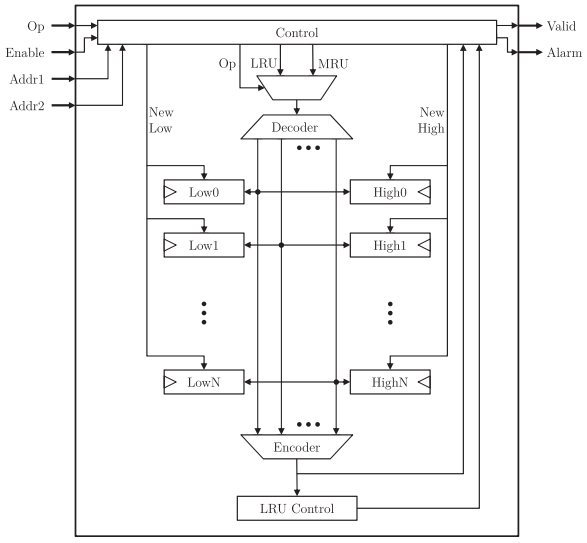

Fig. 10. Range cache hardware for update operations.

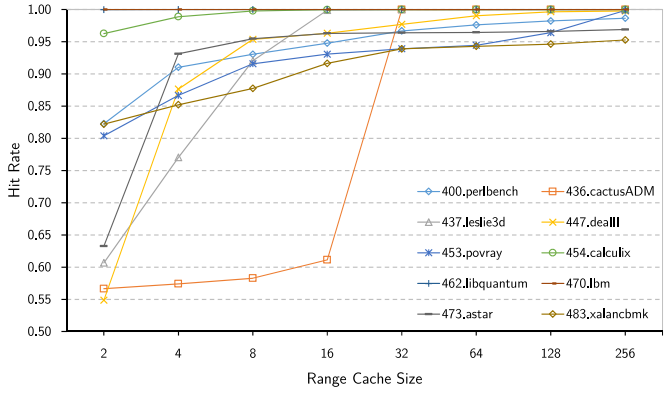

Fig. 11. L1 range cache hit rates for SPEC benchmarks.

Our range cache architecture is similar to a content addressable memory (CAM) in that every entry is searched in a fully associative manner [30]. However, the range cache uses *greater than or equal to* and *less than or equal to* comparison logic instead of *equal to* comparisons used in CAMs. The range cache also requires an additional AND gate for every lower and upper address pair to detect a range cache hit. Finally, CAMs can output a variable length list of storage addresses or associated values for a given input key. Our range cache architecture only outputs a single bit indicating whether the input address falls within a range in the cache.

CAM implementations can be found in many modern computing devices that require fast memory accesses. For example, network routers and switches use CAMs to perform MAC address and IP address forwarding at physical link speed and can store hundreds of thousands of IPv4 and IPv6 routes [31]. In addition, fully-associative translation lookaside buffers (TLBs) are often implemented as CAMs, such as in the ARM Cortex-A9 [32] processor. However, TLBs are much smaller than hardware routing tables and usually contain 512 or fewer entries. As shown in the following section, the maximum range cache size we propose is only 256 entries which is comparable with typical TLB sizes. Therefore, we believe a CAM-based implementation of our range cache architecture is both practical and scalable.

3.4.3 Range Cache Configuration Experiments

After designing the range cache hardware architecture, we analyzed the effect of different cache configurations on the range cache hit rate using benchmarks and real-world application workloads. Our Pin prototype was extended to simulate the range cache functionality, including search, insert, update, and remove operations. We selected the MRU-based pseudo-LRU cache replacement algorithm to replace range entries in the cache because it can outperform the standard LRU algorithm while requiring less hardware resources [33]. Using our extended prototype, we measured the hit rates of different range cache configurations using SPEC 2006 benchmarks and real-world application workloads. A subset of SPEC benchmarks were used to reduce the simulation time required for each experiment to complete, and the benchmarks were selected using [34] as a guide.

Fig. 11 shows the hit rate results for SPEC benchmarks using single level range cache configurations varying in number of range entries. As shown in the figure, hit rates increase until a range cache size of 64 entries, at which point

TABLE 2
Real-World Applications and Workloads

Application	Workload
Firefox	Visit google.com
Evince	Scroll through a PDF
SSH Client	Execute `ls -l` on remote
VLC Media Player	30 seconds of MP3 playback
LibreOffice Calc	View a spreadsheet
LibreOffice Writer	View a word document
LibreOffice Impress	View a presentation

increasing the range cache size does not significantly improve overall hit rates for the benchmarks. However, the hit rate for 453.povray continues to improve as the size of the range cache is increased beyond 64 entries. The sudden increase in hit rate between 16-entry and 32-entry cache sizes for 436.cactusADM suggests the benchmark has a working set size larger than 16 but less than 32 dynamically allocated memory ranges.

We also analyzed range cache performance using the real-world application workloads listed in Table 2, and the experimental results using single level range cache configurations are shown in Fig. 12. All workloads achieve hit rates over 90 percent using just a 32-entry range cache. Unlike SPEC benchmarks, increasing the range cache size steadily improves hit rates to over 98 percent for all real-world application workloads using a 256-entry range cache.

We also estimated the impact of a range cache on overall system performance and found the runtime overhead to be quite low. First, we measured the runtime of a subset of SPEC benchmarks by averaging several runs on a Linux host using a Core2 Quad processor running at 2.0 GHz with 8 GB of memory. Total execution times were collected from the same subset of benchmarks used for our range cache hit rate experiments. Next, we used the miss counts from our single-level range cache experiments to calculate the amount of additional runtime required to service range cache misses for each benchmark. As we have previously mentioned, we expect our range cache to have an implementation similar to a fully associative TLB and therefore select a miss penalty of 20 cycles which is comparable to that of modern TLBs [32]. Assuming a clock speed of 2.0 GHz, we find the total runtime overhead of a 256-entry range cache is only 2.9 percent for the subset of SPEC benchmarks. However, we expect the actual overhead to be lower because our estimation assumes runtime resolution of range cache misses cannot be fully or partially overlapped with any other processor operations, such as resolving data cache misses.

3.5 Armor Synthesis Experiments

To determine the hardware implementation cost of Armor, we augmented the open-source Risc-V Rocket Core with the proposed Armor design [35]. Rocket features an in-order pipeline with branch prediction, virtual memory, and an FPU. The core is part of a system on chip generator written with Chisel.

We added the allocation checking unit to the write-back stage so all register values setup for the function call are committed before the checker finds a matching program counter. Additional read ports were added to the register

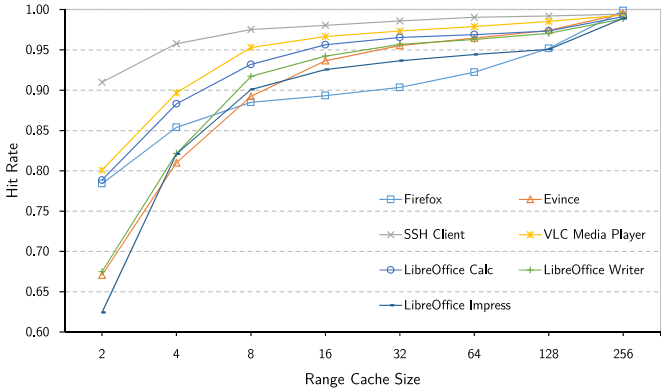

Fig. 12. L1 range cache hit rates for real-world applications.

file to allow Armor to latch needed values for the given function call. Alternatively, required register values could be tracked and recorded via snooping the data used in the write-back stage to remove the need for additional register file read ports. The memory checking unit was added as part of the memory stage, continuously watching incoming virtual memory addresses. When an incoming address is not present in the range cache and not filtered, the unit will raise an exception that will flush the pipeline and trap to the operating system's exception handler.

In addition to Rocket, we also added Armor to the open-source Berkeley Out-of-Order Machine (BOOM) [36]. BOOM features a more modern out-of-order pipeline with register renaming, multiple execution units, and a dedicated load-store unit. We decided to add Armor at the commit stage of BOOM since all instructions are committed in program order. We set up several register tracking mechanisms which monitor committing instruction's logical register destination and record any writes to relevant registers. These mechanisms hold a snapshot of the committed state of the desired registers.

The Load-Store Unit (LSU) operates as its own functional unit. Incoming loads are serviced by memory as soon as possible, meaning memory is read speculatively. Even in this case, the LSU keeps track of the memory addresses in the load queue until the corresponding instruction is committed. This means that Armor's memory range validation can check the load at commit and raise an exception if the address turns out to be invalid. All dependent instructions which wrongly used the loaded data are flushed and the processor will trap to the exception handler. Stores are not sent to memory until commit and the determination of address validity can be done in the same way as loads.

We used the Rocket Chip SoC generator to create a Verilog implementation of our designs. We then synthesized them with the Cadence RTL Compiler using the NanGate 15 nm Open Cell Library [37]. Table 3 shows the synthesis results for our designs when targeting a 1 GHz clock frequency.

For the Rocket core, we observed a modest 1.08x increase in both area and power requirements. The largest contributor to the increase is the 256-entry range cache, accounting for 6.6 percent of the total area and 1.1 percent of the power. In both the unmodified and modified Rocket core, the critical path was observed in the floating point unit, giving about 30 ps of slack.

TABLE 3
Synthesis Results of Rocket and BOOM Augmented with Armor

Core	Area (μm^2)	Slack (ps)	Power (mW)
Rocket	479,960	29	153.0
Rocket+Armor	518,175	30	165.4
Range Cache	34,015	-	1.8
Boom	1,060,592	1	476.3
Boom+Armor	1,173,302	1	524.6
Range Cache	92,543	-	5.7

Values reported are estimates from RTL compiler. Time slack is given for a clock frequency of 1 GHz *and is only reported for overall critical path.*

For the BOOM core, we observed a 1.1x increase in area and power. Again, the range cache accounted for most of the increase. The range cache implemented in the BOOM core required additional check ports to handle multiple memory commits in the same cycle. This introduced duplicate range checking hardware that increased the overall gate count and power consumption compared to the implementation for Rocket. Relative to the size of BOOM, the range cache only took 7.9 percent of the core area and 1.1 percent of the total power. Similar to Rocket, our synthesized BOOM design observed its critical path in the floating point unit, leaving about 1 ps of slack in both the unmodified and modified design.

4 Implementation Considerations

This section briefly discusses some of the implementation considerations we identified during testing that would need to be carefully examined when integrating our monitoring approach into a real system.

First, additional operating system support is needed for our hardware to monitor multiple processes. Target registers in the memory allocation tracking modules, filtering registers in the memory checking hardware, and other configuration registers must be saved during a context switch. Currently, Linux's *task_struct* contains between 1,000-2,000 bytes of data. To support the Armor configuration, approximately 88 bytes of information will need to be stored in this struct and reloaded during a context switch. In addition, the operating system could store the 256 range cache entries, resulting in an additional 4,096 bytes of information to store. Process identification values should also be assigned to range cache entries to associate an entry with the correct application. These additional requirements are necessary to support monitoring multiple programs and have the potential to add extra latency when performing context switches and range cache operations.

Multithreaded applications running on our monitoring hardware may require additional software support depending on the threading library implementation. Our experiments utilized the pthreads library, which uses `mmap` to dynamically allocate a region of memory for a thread's stack. This region is detected by the memory allocation tracking hardware, and every subsequent memory access made to the thread's stack is verified by querying the range cache. Although this behavior does not generate false positives, it may create a performance bottleneck when checking every stack memory access. To avoid this pitfall, the base address of the thread's stack can be written into the stack base filtering register, effectively eliminating range cache searches when the thread's stack memory is accessed.

Reserved stack regions that do not lie between the stack base address and the stack pointer also require modifications to our hardware design. For example, the red zone is a 128 byte region located below the stack pointer that a function can use as temporary storage and is mandated by the System V AMD64 ABI. We observed false positive memory access violations when applications accessed memory in the red zone because the memory addresses are not filtered and do not belong to any tracked dynamically allocated memory regions. Additional hardware is needed that subtracts the size of the red zone from the current stack pointer before performing address range filtering.

Many operating systems place a limit on an application's stack size. False positives triggered by accessing reserved stack memory regions, such as the red zone, can be eliminated by using the stack size limit to determine the end address of the stack instead of using the stack pointer register to filter stack memory accesses. However, stack overflows that occur between the stack pointer and stack limit address will no longer be detected by our monitoring hardware. Furthermore, some operating systems support unlimited application stack sizes by implementing a linked list of several smaller stack memory regions. Additional filtering registers are needed to filter all contiguous stack memory regions in the list, and the operating system must write these registers when a new stack region is added to the linked list.

In addition to stack filtering considerations, monitoring certain memory allocation functions also presents challenges. The virtual addresses of custom or non-standard memory allocation functions must be located manually and supplied to the program loader. Hardware configuration options for custom functions, such as the arguments to capture, must also be communicated to the program loader. Functions that break a contiguous memory region into two separate regions, such as `munmap`, introduce several new options for a range cache implementation. A hardware design may elect to place the address pairs for both ranges into the cache, or only one address pair may be inserted into the range cache. If one address range is placed into the cache, selecting between the two pairs is a design choice that may impact the hit rate and warrants further experimental analysis.

Configuring Armor for a specific ABI would limit its supported operating systems. However, ABIs in common use today, such as AArch64, Risc-V and System V, feature a call convention. To enable Armor to be convention-agnostic, a function table that includes additional software configured metadata in addition to the function's start addressed can be added. This table would include information about which register or which stack offset where relevant arguments are stored, and an opcode indicating what type of function it is (malloc, calloc, etc...). Since Armor has read access to the instruction stream and architectural registers, acquiring a register can be configured via software at runtime. Methods discussed previously can be leveraged to extract arguments from stack locations.

While testing our Pin prototype, we observed that jump instructions were periodically used to return from allocation functions instead of explicit return instructions.

Support for this type of execution behavior requires the monitoring hardware to determine the allocation function's return address by reading the address in the processor's link register and also using the address in the instruction pointer register. While the monitor waits for the allocation function to complete, the instruction pointer should be compared with both the saved return address from the link register and the saved return address computed using the value in the instruction pointer register. No changes are needed to capture the return value from the function because the processor's calling convention is still followed.

Additional hardware modifications are needed when implementing our monitor alongside a processor that passes allocation function arguments on the stack. The most straightforward solution is for the processor to communicate the function allocation arguments to the monitor directly. Another potential solution is implementing a hardware shift register that stores several of the most recent values pushed onto the stack, and then reading allocation arguments from the shift register. The monitor could also delay capturing function arguments until the beginning of the allocation routine provided that the arguments are always popped off the stack into specific registers.

Many modern processors are capable of loading multiple bytes from memory with a single instruction. As a result, multiple address checks are necessary for each memory instruction that accesses multiple bytes in a dynamically allocated memory region. The memory checking hardware may need to output a size parameter or a lower and upper address pair to allow the range cache to check all accessed addresses.

Finally, systems that do not have memory page protections can use our monitoring approach to protect the integrity of application binaries with only a slight modification. Removing the memory addresses of an application's text section from the hardware filtering registers enables the monitor to detect attacks targeting the program's instructions. We note that loading an application's instructions from memory will not cause our monitor to produce false positive violations because instruction fetches do not use explicit processor instructions to perform the load from memory.

5 Related Work

In this section, we review previous hardware and software-based memory safety approaches. We also discuss previous metadata caches related to our architecture.

5.1 Memory Safety

Previous memory safety work focuses on detection and prevention of spatial and temporal memory errors. Spatial memory errors occur when a pointer accesses memory beyond the bounds of the object to which it points, and temporal memory errors occur when a program uses uninitialized memory or accesses memory that has already been deallocated.

Tripwire approaches enforce spatial safety by surrounding allocated objects in memory with special bytes that generate exceptions when accessed. Software implementations of tripwires incur high runtime overheads, reaching 20x slowdown in some cases [38]. SafeMem [38] and MemTracker [14] implement tripwires in hardware to reduce runtime overhead to less than 5 percent. Although tripwires are an efficient solution for enforcing spatial safety, an attacker can bypass them by simply incrementing a pointer past the tripwire bytes before dereferencing the pointer.

Coloring techniques implement spatial safety by tagging pointers and their respective memory regions with unique identifiers called colors. A dereferenced pointer's color must be the same as the color of the memory region to which it points. Previous hardware implementations incur 10 percent runtime overheads [15]. The primary drawback to coloring approaches is that the spatial protection is probabilistic. With only a finite amount of unique colors, there is always a chance that multiple memory regions will share the same color. Spatial violations involving a pointer that moves between memory regions with the same color will remain undetected.

Fat pointers provide spatial memory protection by extending pointer representation to include the lower and upper address information for the memory segment. When the pointer is dereferenced, it is verified to be within the bounds specified by the lower and upper addresses. Jim et al. propose a new programming language, Cyclone, that incorporates the use of fat pointers to ensure spatial memory safety [13]. Milewicz et al. retrofit existing source code with fat pointer representation and runtime checking of pointers [8].

Hardware implementations of fat pointers aim to further reduce the performance and memory costs. Watchdog [16] uses micro-operations to query a hardware bounds table, resulting in 15-24 percent runtime overhead on average. Intel Memory Protection Extensions (MPX) provide fat pointer support starting with the Skylake microarchitecture [3]. Fat pointers are an attractive solution because, unlike tripwires and coloring, fat pointers provide complete spatial memory protection. However, because they change the representation of a pointer, fat pointers require recompilation of all source code used by an application. To address the compatibility challenges of fat pointers, object bounds approaches trade a small amount of spatial safety for compatibility by storing memory bounds of an object in a disjoint memory space. Pointer arithmetic operations are checked against bounds information to ensure pointers point to their intended object.

AHEMS [18] implements object bounds checking in hardware using an asynchronous coprocessor architecture. Runtime overhead is only 10 percent on average, but the asynchronous processing of object bounds creates a window of time between a spatial violation and its detection that may be large enough to carry out an attack. Although object bounds techniques are compatible with unsupported or third-party code and libraries, the approach does not provide full spatial protection because memory inside objects (such as sub-objects, arrays, or structures) cannot be protected.

Many software techniques have been proposed to enforce temporal memory safety. Memcheck [12] implements temporal safety using the Valgrind dynamic binary instrumentation framework. AddressSanitizer [9] detects use-after-free vulnerabilities at just 73 percent runtime cost but requires recompilation of source code. MemorySanitizer [10] uses static compiler instrumentation to check for use of uninitialized memory and has 2.5x runtime overhead.

Our monitoring approach provides temporal safety as well as spatial safety at object bounds granularity, and

unlike all previous work, our technique does not require the monitored application to be recompiled or instrumented.

5.2 Metadata Caches

Metadata processing architectures associate configurable metadata tags with various components of program execution to enforce different security, debug, or performance policies. Similar to data caches, metadata tags are often stored in caches to improve performance.

Venkataramani et al. [14] analyze different metadata cache configurations and find a 2 KB two-way set-associative dedicated cache provides the best balance of performance and area cost [14]. Other work has examined the effect of shared cache configurations in lower level caches. Venkataramani et al. find a dedicated first level metadata cache and shared lower level caches have negligible impact on performance [27]. The same metadata cache configurations are used in [15].

Several unconventional cache architectures have also been proposed. Similar to classical data caches, previous metadata cache architectures use addresses to lookup cached metadata tags. Harmoni [19] uses instruction opcodes to lookup cached metadata tags. Dhawan et al. implement metadata tag compression to better utilize cache space [20].

Tiwari et al. assign metadata tags to address ranges. The first level cache is used to determine which range an address exists in, and the second level cache is used to access the metadata tag associated with the range. Range coalescing is performed in hardware to maximize the effective cache capacity [21]. Our caching architecture stores metadata in a range pair format similar to Tiwari et al. but does not perform range coalescing.

6 Conclusion

In this paper, we introduced an architectural approach for tracking dynamic memory allocations and securing heap memory accesses that does not require recompilation of source code or instrumentation of the target application. We showed how our hardware leverages a processor's calling convention to track memory allocations and validate memory accesses at runtime. A software prototype of our approach detected 98 percent of memory errors in security test cases from the NSA Juliet Test Suite and two real-world exploits, demonstrating our architecture is effective in practice. Simulation experiments of our range cache architecture using SPEC 2006 benchmarks and real-world workloads produced hit rates over 95 percent for a 256-entry range cache, and the additional runtime overhead was only 2.9 percent.

Directions for future work include additional range cache analysis, including measuring worst case performance overheads using different range and data cache configurations, and experimenting with different multi-level range cache designs to obtain higher hit rates. Support for uninitialized memory checking can be added to the monitoring hardware to detect a wider range of temporal memory errors. Future work might also explore different filtering approaches that support non-contiguous stack memory used by multithreaded applications and systems that support unlimited stack sizes. Finally, constructing a full system prototype in reconfigurable fabric may reveal additional implementation details not identified in this work.

References

[1] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. CMU-CS-02–197, 2002.

[2] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. USENIX Security Symp.*, 1998, pp. 63–78.

[3] R. Ramakesavan, D. Zimmerman, P. Singaravelu, G. Kuan, B. Vajda, S. Gibbons, and G. Beeraka, "Intel memory protection extensions enabling guide," 2016. [Online]. Available: http://software.intel.com

[4] T. Wei, T. Wang, L. Duan, and J. Luo, "Secure dynamic code generation against spraying," in *Proc. ACM Conf. Comput. Commun. Security*, 2010, pp. 738–740.

[5] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. ACM Conf. Comput. Commun. Security*, 2004, pp. 298–307.

[6] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. ACM Conf. Comput. Commun. Security*, 2007, pp. 552–561.

[7] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inform. System Security*, vol. 15, no. 1, pp. 2:1–2:34, 2012.

[8] R. Milewicz, R. Vanka, J. Tuck, D. Quinlan, and P. Pirkelbauer, "Runtime checking C programs," in *Proc. Symp. Appl. Comput.*, 2015, pp. 2107–2114.

[9] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 309–318.

[10] E. Stepanov and K. Serebryany, "MemorySanitizer: Fast detector of uninitialized memory use in C++," in *Proc. IEEE/ACM Int. Symp. Code Generation Optim*, 2015, pp. 46–55.

[11] S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," in *Proc. Eur. Softw. Eng. Conf. Held Jointly ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2003, pp. 307–316.

[12] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 17–30.

[13] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. USENIX Annu. Tech. Conf.*, 2002, pp. 275–288.

[14] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: Efficient and programmable support for memory access monitoring and debugging," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2007, pp. 273–284.

[15] I. Doudalis, J. Clause, G. Venkataramani, M. Prvulovic, and A. Orso, "Effective and efficient memory protection using dynamic tainting," *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 87–100, Jan. 2012.

[16] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proc. Int. Symp. Comput. Archit.*, 2012, pp. 189–200.

[17] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proc. ACM Conf. Comput Commun. Security*, 2013, pp. 721–732.

[18] K.-Y. Tseng, D. Lu, Z. Kalbarczyk, and R. Iyer, "AHEMS: Asynchronous hardware-enforced memory safety," in *Proc. Euromicro Conf. Digital Syst. Des.*, 2014, pp. 183–190.

[19] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2012, pp. 1–12.

[20] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 487–502.

[21] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood, "A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2008, pp. 94–105.
[22] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *Proc. USENIX Security Symp.*, 2014, pp. 845–860.
[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2005, pp. 190–200.
[24] National Security Agency Center for Assured Software, "Juliet test suite v1.2 for C/C++." (2016). [Online]. Available: http://samate.nist.gov
[25] CVE-2014–0160, Available from MITRE, CVE-ID CVE-2014–0160., Apr. 2014. [Online]. Available: http://cve.mitre.org/
[26] CVE-2015–0235, Available from MITRE, CVE-ID CVE-2015–0235., Jan. 2015. [Online]. Available: http://cve.mitre.org/
[27] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2008, pp. 173–184.
[28] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.
[29] Y.-H. E. Yang and V. K. Prasanna, "High throughput and large capacity pipelined dynamic search tree on FPGA," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2010, pp. 83–92.
[30] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.
[31] Cisco Systems, Inc., *CAT 6500 and 7600 Series Routers and Switches TCAM Allocation Adjustment Procedures*, Aug. 2014.
[32] ARM Limited, *ARM Cortex-A9 Tech. Reference Manual*, 2016, revision: r4p1.
[33] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite," in *Proc. Southeast Regional Conf.*, 2004, pp. 267–272.
[34] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *Proc. Int. Symp. Comput. Archit.*, 2007, pp. 412–423.
[35] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016–17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[36] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," EECS Department, University of California, Berkeley CA, USA, Tech. Rep. UCB/EECS-2015–167, Jun. 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html
[37] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm FreePDK technology," in *Proc. Symp. Int. Symp. Phys. Des.*, 2015, pp. 171–178.
[38] F. Qin, S. Lu, and Y. Zhou, "SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2005, pp. 291–302.

Alex Grieve received the BS and MS degrees in computer engineering from Iowa State University, in 2014 and 2016, respectively. He is currently with RFA Engineering working as a robotics and software engineer at John Deere Intelligent Solutions Group. His research interests include computer architecture, compilers, software protection and security, and embedded systems.

Michael Davies is currently working toward the undergraduate degree in computer engineering and mathematics with Iowa State University. Upon graduation, he plans to attend graduate school to pursue a PhD in electrical and computer engineering. His research interests include computer architecture, reconfigurable computing, machine simulation, and high-performance computing.

Phillip H. Jones received the BS and MS degrees in electrical engineering from the University of Illinois at Urbana-Champaign, in 1999 and 2002, and the PhD degree in computer engineering from Washington University in St. Louis, in 2008. Currently, he is an associate professor in the Department of Electrical and Computer Engineering, Iowa State University, Ames, where he has been since 2008. His research interests include adaptive computing systems, reconfigurable hardware, embedded systems, and hardware architectures for application-specific acceleration. He is a member of the IEEE.

Joseph Zambreno received the BS (summa cum laude) degree in computer engineering, the MS degree in electrical and computer engineering, and the PhD degree in electrical and computer engineering, from Northwestern University in Evanston, Illinois, in 2001, 2002, and 2006. He has been with the Department of Electrical and Computer Engineering, Iowa State University since 2006, where he is currently a professor. His research interests include computer architecture, compilers, embedded systems, reconfigurable computing, and hardware/software co-design, with a focus on run-time reconfigurable architectures and compiler techniques for software protection. He is a senior member of the IEEE.