2016

A recompilation and instrumentation-free monitoring architecture for detecting heap memory errors and exploits

Alex Ross Grieve
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Engineering Commons

Recommended Citation
Grieve, Alex Ross, "A recompilation and instrumentation-free monitoring architecture for detecting heap memory errors and exploits" (2016). Graduate Theses and Dissertations. 15921.
https://lib.dr.iastate.edu/etd/15921

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
A recompilation and instrumentation-free monitoring architecture for detecting heap memory errors and exploits

by

Alex Ross Grieve

A thesis submitted to the graduate faculty in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor
Phillip H. Jones
Doug W. Jacobson

Iowa State University
Ames, Iowa
2016

Copyright © Alex Ross Grieve, 2016. All rights reserved.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>List of Tables</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>Abstract</td>
<td>vi</td>
</tr>
<tr>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2. Background: Program Anatomy and Execution Behavior</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Program Startup and Memory Layout</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Shared Libraries and Dynamic Linking</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Dynamic Memory Allocation</td>
<td>8</td>
</tr>
<tr>
<td>Chapter 3. Memory Protection Approach</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Constraints, Requirements, and Assumptions</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Dynamic Memory Allocation Tracking</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Validating Memory Accesses</td>
<td>14</td>
</tr>
<tr>
<td>Chapter 4. Software Prototype and Security Analysis</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Software Prototype</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Security Benchmark Testing</td>
<td>21</td>
</tr>
<tr>
<td>4.3 Real-World Exploit Testing</td>
<td>24</td>
</tr>
<tr>
<td>4.3.1 CVE-2014-0160 (Heartbleed)</td>
<td>25</td>
</tr>
<tr>
<td>4.3.2 CVE-2015-0235 (GHOST)</td>
<td>26</td>
</tr>
<tr>
<td>4.3.3 CVE-2016-4324 (LibreOffice)</td>
<td>26</td>
</tr>
<tr>
<td>4.4 Limitations</td>
<td>27</td>
</tr>
</tbody>
</table>
CHAPTER 5. HARDWARE ARCHITECTURE 

5.1 Top Level Design 

5.2 Memory Allocation Tracking Architecture 

5.3 Memory Checking Architecture 

5.4 Allocated Memory Storage Architecture 

5.4.1 Storage Format 

5.4.2 Range Cache Architecture 

5.4.3 Range Cache Configuration Experiments 

5.5 Implementation Considerations 

CHAPTER 6. RELATED WORK 

6.1 Probabilistic Approaches 

6.2 Integrity Mechanisms 

6.3 Memory Safety 

6.3.1 Spatial Safety 

6.3.2 Temporal Safety 

6.4 Metadata Caches 

CHAPTER 7. CONCLUSION 

BIBLIOGRAPHY
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Software prototype test results using the Juliet Test Suite</td>
<td>22</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Memory overhead of metadata bit and range pair storage formats for SPEC 2006 integer benchmarks</td>
<td>35</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Memory overhead of metadata bit and range pair storage formats for SPEC 2006 floating point benchmarks</td>
<td>37</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Real-world applications and workloads</td>
<td>43</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>2.1</td>
<td>Example layout of a program’s ELF segments in memory</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Example usage of the Global Offset Table (GOT) to locate a shared library’s private data segment at runtime</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Execution sequence of lazy function resolution by the dynamic linker</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>High level state machine depicting our dynamic memory allocation tracking approach</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>High level state machine illustrating our memory checking approach</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Memory accesses to filtered regions are not validated</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Code illustrating the core functionality of a memory allocation wrapper function used in the software prototype</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Code depicting the main functionality of the memory checking function used in the software prototype</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Code snippet of the undetected heap-based buffer overflow test case</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Diagram of the top level design of our hardware architecture</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Hardware architecture of a dynamic memory allocation tracking submodule</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Hardware architecture of a submodule that outputs memory addresses for checking</td>
<td>33</td>
</tr>
<tr>
<td>5.4</td>
<td>Memory overhead of the range pair storage format normalized to metadata bit storage overhead for SPEC 2006 integer benchmarks</td>
<td>36</td>
</tr>
</tbody>
</table>
Figure 5.5  Memory overhead of the range pair storage format normalized to metadata bit storage overhead for SPEC 2006 floating point benchmarks . . 38
Figure 5.6  Range cache hardware for search operations . . . . . . . . . . . . . . . . 39
Figure 5.7  Range cache hardware for update operations . . . . . . . . . . . . . . . . 40
Figure 5.8  Single level range cache hit rates for SPEC benchmarks . . . . . . . . . 42
Figure 5.9  Single level range cache hit rates for real-world applications . . . . . . 43
Figure 5.10 L2 range cache hit rates for real-world applications . . . . . . . . . . . . 44
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 work 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. Security analysis using a software prototype shows our architecture detects 98% of heap memory errors from selected test cases in the Juliet Test Suite and real-world exploits. Performance simulations of our architecture using SPEC benchmarks and real-world application workloads show our architecture achieves hit rates over 95% for a 256-entry cache, resulting in only 2.9% runtime overhead.
CHAPTER 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 [66, 21, 27, 37, 8, 90, 71]. However, current protection mechanisms can still be circumvented as shown by recent real-world exploits [94, 14, 62, 79, 95, 78, 88, 73].

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 [55], augmenting compilers to insert checks at compile time [40, 57, 47, 74, 34, 6, 99, 76, 82, 58], or instrumenting the application binary to perform memory access checking at runtime [98, 77, 17]. Several new, memory safe programming languages based on C and C++ have also been proposed [46, 60]. 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 [93, 25, 30, 35, 32, 38, 56, 52, 96, 89, 31, 33, 87]. 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 thesis 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 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% are achievable for a 256-entry cache, resulting in just 2.9% runtime overhead. Security experiments using test cases from the NSA Juliet Test Suite and real-world exploits show our architecture successfully detects 98% of heap memory errors spanning eight different classes as well as one type of stack memory error.

The remainder of this thesis is organized as follows. Chapter 2 explains program anatomy and execution behaviors, including the startup process, dynamic linking, and dynamic memory allocation. Chapter 3 introduces our heap memory protection approach, and Chapter 4 analyzes its security effectiveness using benchmarks and real-world exploits. Chapter 5 describes the hardware architecture that implements our heap memory protection approach, and its performance is analyzed using SPEC benchmarks and real-world application workloads. Chapter 6 reviews related work in the area of C and C++ software protection approaches. Chapter 7 offers our conclusion and outlines future work.
CHAPTER 2. BACKGROUND: PROGRAM ANATOMY AND EXECUTION BEHAVIOR

This chapter contains background information necessary for understanding the security implications of program execution. The program startup process and general layout of a program in memory is illustrated, and shared libraries and dynamic linking are discussed. System calls used to dynamically allocate additional memory at runtime are examined. The chapter concludes with a description of library functions commonly used by applications to obtain memory from the system at runtime. We note that the discussion will be in the context of Linux due to its open-source nature, but the same concepts are generally applicable to other operating systems.

2.1 Program Startup and Memory Layout

The program startup process begins with the operating system kernel handling an `execve` system call. The kernel begins parsing the Executable and Linkable Format (ELF) header to locate the ELF Program Header. The ELF file format is the standard binary file format for executables in Unix environments as well as object files and shared libraries [86]. The ELF Program Header contains all of the program’s segments and their respective memory access permissions. The kernel sets up the program’s virtual address space by loading segments labeled as `PT_LOAD` into memory and also allocates memory for the program’s stack and heap.

Figure 2.1 illustrates an example layout of a program in memory after the kernel has mapped its segments into the virtual address space. The text segment resides on executable memory pages and contains the application’s machine instructions to be executed by the processor. The data segment contains static and global variables that have been initialized by the programmer.
to a non-zero value. The bss segment consists of uninitialized static and global variables (or those initialized to zero) and is filled with zeros. Both the data and bss segments are loaded onto non-executable memory pages and are often adjacent to each other in memory. Note that there many other segments not pictured that are necessary to support other facilities such as position independent code and multithreading.

Once the kernel has mapped all of the executable’s ELF segments into memory, it then allocates memory for the program’s stack and adjusts the program break to point to the start of the heap. The kernel checks the ELF Program Header for a PT_INTERP entry, and if none is found, the executable is considered static. Static programs resolve library functions, external variable addresses, and other dependencies at compile-time, and contain copies of source code from libraries they utilize. The kernel passes arguments to the program by placing them on the

Figure 2.1: Example layout of a program’s ELF segments in memory
stack, sets the instruction pointer register to the entry point address specified in the program’s ELF Program Header, and control is transferred to the program.

2.2 Shared Libraries and Dynamic Linking

Most modern executables utilize dynamic linking, where the name of shared libraries and other dependencies are placed in the program’s ELF headers as relocation symbols at compile time. At runtime, the dependencies are loaded into the program’s address space in the memory mapping segment, as illustrated in Figure 2.1, and the addresses of all relocation symbols are resolved.

Dynamically linked applications have several advantages over their statically compiled counterparts. Dynamically linked programs are smaller in size than static programs because library source code is not compiled into the final executable. Multiple applications can share one physical copy of a library’s text segment and only need their own private copy of data segments, thereby reducing the physical memory footprint of the program. Additionally, dynamically linked executables can make use of newer library versions without the need for recompilation.

Dynamically linked executables incur a small performance overhead when locating shared libraries and performing dependency resolution. To perform these operations, the kernel loads the dynamic linker specified by the PT_INTERP ELF entry into the program’s memory mapping segment. On most Linux systems, the dynamic linker is a small, statically compiled program called ld-linux.so. The kernel places an auxiliary vector of information on the stack that contains important information about the program, such as its memory location and its entry point, needed by the dynamic linker to do its job. Control is then passed to the dynamic linker.

The dynamic linker parses the ELF Program header for dynamic dependencies, locates the dependencies, and maps them into the program’s memory mapping segment. Dependencies are typically mapped as private copies so that relocations do not get written to disk and break other applications using the shared dependency. Once all dependencies have been loaded into the program’s address space, the dynamic linker proceeds to resolve symbol relocations by determining the runtime addresses of static and global data as well as shared library functions.
As previously mentioned, the text segment is the only portion of a shared library that is actually shared between processes. Each process that uses a shared library gets its own exclusive copy of other segments. A problem that arises from this arrangement is that there is no way for the shared library code to know the runtime address of its private data segment. The solution to this problem is a special ELF segment called the Global Offset Table (GOT.) The GOT is a table of addresses in the data section that contains an entry for each variable in the data segment. The GOT is always placed at a known offset from the shared library’s text section when loaded into a program’s memory mapped segment. Instead of using absolute addressing to reference a variable in the data segment, the shared library code references the variable’s corresponding entry in the GOT as shown in Figure 2.2. At runtime, the dynamic linker knows the location of the program’s copy of the data segment and populates each GOT entry with the correct absolute address. The GOT provides a level of indirection that allows the shared library code to function correctly regardless of where the private data segment is loaded.

The Procedure Linkage Table (PLT) is used in conjunction with the GOT to resolve the addresses of shared library functions at runtime. Each function whose address cannot be deter-
mined at compile time gets an entry in the PLT and the GOT. Additionally, there is a special entry in the PLT that invokes the dynamic linker to resolve function addresses at runtime. Instead of calling a function using an absolute address, the program calls the corresponding entry in the PLT. The PLT entry first jumps to its corresponding GOT entry, which initially points back to the PLT entry as illustrated by steps 1 and 2 in Figure 2.2a. The PLT entry contains instructions that set up the appropriate registers with arguments for the dynamic linker to resolve the function address, and then invokes the dynamic linker by jumping to the special PLT entry shown in step 3 of Figure 2.2a. The dynamic linker writes the address of the function into the correct GOT entry, and then the program resumes normal execution. The next time the function is called, the dynamic linker will not be invoked because the correct address of the function is in the GOT depicted in Figure 2.2b. The dynamic linker can be configured to resolve all function addresses before program execution begins or in a lazy, as-needed fashion [36].
2.3 Dynamic Memory Allocation

There are a variety of reasons why a program may need to allocate memory dynamically on the heap. For example, a program’s memory requirements may be dependent on a variable whose value cannot be determined at compile time. Allocating an object on the heap also affords the programmer greater control over the object’s lifecycle compared to stack allocation. Often, large or variable length blocks of memory are allocated on the heap to conserve the relatively limited amount of stack memory.

The `brk` system call manipulates the address of the program break and is the traditional method of dynamic memory allocation in Linux systems [26]. As illustrated in Figure 2.1, the program break defines the top of the heap and typically starts at the end of the data segment. Manipulating the program break with the `brk` system call has the effect of dynamically growing or shrinking the heap memory allocated to the program. We note that the program break only grows in one direction and is always contiguous with respect to its starting position. Although this limitation makes the `brk` system call implementation efficient, its use often results in internal fragmentation because only memory located at the very top of the heap is able to be reclaimed by the system [44].

Another system call that programs can use to obtain more memory at runtime is the `mmap` system call [26]. The `mmap` system call allocates memory by mapping an independent region of memory into the program’s virtual address space. The memory region can be configured as either shared or private as well as file or anonymous mapped [1]. Changes to shared memory mapped regions are committed to the underlying file, making them useful for inter-process communication, whereas private memory mapped regions do not have changes committed to any underlying file. File mappings load files from disk into memory and are used during program startup. Anonymous mappings have no file on disk backing the memory region, and the memory region is zeroed by the kernel before it is mapped into the program’s address space. To obtain more heap memory, a program will typically invoke the `mmap` system call with the private and anonymous arguments.
Unlike `brk`, memory allocated with `mmap` is not restricted to a single contiguous block of memory. As a result, large allocations that fail using `brk` will likely succeed with `mmap`. Internal fragmentation is also reduced, allowing a program to better manage its dynamically allocated memory segments. However, the performance of `mmap` tends to be worse than `brk` because `mmap` writes zeros to the memory before releasing it to the program [1].

Due to the high overhead associated with system calls, programs do not typically invoke these calls directly [44]. Instead, programs use library functions to obtain additional heap memory at runtime. Library functions, such as `malloc` and `free`, perform the necessary system calls and are highly optimized for performance and efficiency.

The GNU C library uses the `ptmalloc` implementation as its dynamic memory allocation library [39]. The `ptmalloc` implementation maintains two core data structures: the allocation list and the free list. The allocation list is a linked list of memory segments allocated to the program, and the free list is a linked list of all available segments of memory obtained from the kernel or deallocated by the program.

When a program calls `malloc`, the free list is searched for the first available block of memory that satisfies the request. The block is moved to the allocation list, and a pointer to the allocated block is returned to the program. Conversely, when the program returns a segment of dynamically allocated memory, it calls `free` with a pointer to the beginning of the dynamically allocated segment. The allocation list is traversed until the allocated segment is found. The block is then removed from the allocation list and placed into the free list.

When the free list is empty or cannot fulfill a certain sized request, `malloc` requests more memory from the kernel by first invoking the `brk` system call. If `brk` fails or the requested allocation is larger than a configurable threshold, `malloc` then invokes the `mmap` system call. It’s worth noting that the `mmap` system call will request much more memory than the program originally asked for with its call to `malloc`. Because system calls are expensive in terms of performance, requesting extra memory from the kernel allows `malloc` to potentially serve future memory requests without making any system calls. Finally, when the free list becomes too large, segments of memory are returned to the kernel by calling the `brk` or `munmap` system calls, depending on how the segment was originally obtained.
CHAPTER 3. MEMORY PROTECTION APPROACH

This chapter introduces the two components of our memory protection approach. First, problem constraints are restated and requirements for our memory protection approach are discussed. Next, we describe our approach for tracking dynamic memory allocations made by an application during runtime. We demonstrate how to leverage the calling convention of a processor to determine the size of the memory allocation and present an adaptation to track dynamic memory allocations made using system calls. Finally, we explain our memory access checking technique and detail our method for filtering memory accesses to statically allocated memory regions.

3.1 Constraints, Requirements, 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 architectural register file and instruction stream. 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` described in Chapter 2.3, 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 [45, 12, 70].

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.

### 3.2 Dynamic Memory Allocation Tracking

The high level state machine in Figure 3.1 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 [45, 12, 70]. When the monitor receives the allocation function’s
Inputs: FuncAddr, IP, RetAddr, Arg0, RetVal
Outputs: LowAddr, HighAddr
Registers: target, size, ret_addr

Figure 3.1: High level state machine depicting our dynamic memory allocation tracking approach

virtual address, the address is stored in the monitor’s target register, and the monitor moves into state S1.

In state S1, 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 [10], the A0 register on 64-bit RISC-V processors [72], and the RDI register for 64-bit x86 processors using the System V AMD64 ABI [43]. The monitor reads the size 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 [10] or the RA register on 64-bit RISC-V processors [72], and saves the address into its ret_addr register. Then, the monitor moves into state S2.

In state S2, 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 [10], the A0 register on 64-bit RISC-V processors [72], and the EAX register for 64-bit x86 processors using the System V AMD64 ABI [43]. 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 Figure 3.1, 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 Figure 3.1, 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 \textit{FuncAddr} input in Figure 3.1 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.

### 3.3 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 Figure 3.2. 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 \textit{opcode} 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
Inputs: MemOp, Ins, MemAddr
Outputs: Alarm
Registers: opcode, addr, valid

Figure 3.2: High level state machine illustrating our memory checking approach

stored in the monitor’s opcode 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 S2, 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 S1.
Otherwise, a heap memory safety violation is detected, and an alarm is raised.

The memory checking approach depicted in Figure 3.2 verifies memory accesses made by an
application at runtime without source code modification, recompilation, or binary instrumenta-
tion. 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 Figure 3.3 are excluded from heap memory access checking.

As shown in Figure 3.3, 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 [8] or Write-XOR-Execute [90] 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. The GOT and PLT 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 ap-
lications, 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 func-
tions. 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.
CHAPTER 4. SOFTWARE PROTOTYPE AND SECURITY ANALYSIS

This chapter introduces our software prototype and the results of our security analysis. The design of the software prototype is discussed, and the security effectiveness of the proposed memory tracking approach is evaluated using benchmarks and real-world exploits. Experimental results are presented, and several limitations of the dynamic memory monitoring approach are identified.

4.1 Software Prototype

We developed a software prototype using Pin to test our dynamic memory allocation tracking and memory checking monitor. Pin is a dynamic binary instrumentation framework used for creating application profiling, computer architecture simulation, security, and other program analysis tools [54]. Our prototype was built using Pin 3.0 and targets 64-bit Linux systems.

To implement the dynamic memory allocation tracking functionality of our monitor, we used the Pin API 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. 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, call the wrapped routine, and record the address ranges of dynamically allocated regions.

The code shown in Figure 4.1 outlines the general design of the wrapper functions using the malloc wrapper function as an example. The global flag variable set in line 7 indicates a memory allocation routine is about to execute and memory accesses should not be checked. The flag also prevents tracking the same dynamically allocated memory region multiple times.
Figure 4.1: Code illustrating the core functionality of a memory allocation wrapper function used in the software prototype

when several allocation function calls are made to service a single allocation request. For example, many calloc implementations allocate memory with malloc, and realloc may invoke malloc or free depending on the arguments. Although comparing the return address of the allocation function with the instruction pointer will prevent an allocated memory region from being tracked multiple times, it requires every application instruction to be instrumented with the check which introduces significant runtime overhead. The global flag provides the same functionally at a fraction of the runtime cost.

The Pin framework is configured to supply each wrapper function with a function pointer to the original routine and all arguments passed to the wrapped function. As shown on line 9 in Figure 4.1, the function pointer is used along with the original arguments to call back to the wrapped routine and capture the function’s return value. After the callback has completed, the global flag is unset, and the return value is inspected for errors. If the dynamic memory allocation completes successfully, the allocated region is identified using the size argument and
the virtual memory address returned by the original routine. Finally, as seen on line 15, a lower and upper address pair is inserted into a binary search tree keyed on the lower address. Wrapper functions for deallocation routines, such as `munmap` or `free`, attempt to remove the address pair from the binary search tree at line 15 and report an error if the pair is not in the tree.

System calls that dynamically allocate memory, specifically `mmap`, `mremap`, `munmap`, and `brk`, are also tracked. Two callback routines are registered using the Pin API that are called directly before and after execution of a system call. The functionality shown in Figure 4.1 is split across both callback routines. The function called directly before the system call checks the system call number, and if a `mmap`, `mremap`, `munmap`, or `brk` system call is set to execute, it stores the appropriate arguments and sets the global flag accordingly. The function called immediately after the system call examines the return value for errors, unsets the global flag, and updates the binary search tree containing the allocated memory range pairs.

```c
/*
 * Memory checking function
 */
void memory_access(void* addr, void* sp) {
    if (!main_running || flag != none)
        return;

    if (addr <= stack_base && addr >= sp)
        return;

    if (IsMainExecutable(addr))
        return;

    if (tree->contains(addr))
        return;

    log_violation(addr);
}
```

Figure 4.2: Code depicting the main functionality of the memory checking function used in the software prototype
To enforce memory safety, we configured the Pin framework to verify all executed memory instructions using the memory checking function shown in Figure 4.2. The code filters memory accesses to filtered regions and verifies the memory address is within the bounds of a valid heap memory region allocated to the application. As shown on line 6, memory checking is suspended outside of the program’s main function and inside dynamic memory allocation routines. Memory accesses to the program’s stack and data segments are also filtered on lines 9 and 12, respectively. The binary search tree containing the lower and upper address pairs of dynamically allocated memory regions is searched on line 15. If the memory address does not exist in the tree, a memory safety violation is detected and written to a log file.

4.2 Security Benchmark Testing

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++. Although the Juliet Test Suite contains over 60,000 test cases spanning 118 different Common Weakness Enumerations (CWEs) [59], we limit our experiments to only those cases containing memory errors involving dynamically allocated memory regions. Each test case contains several different source code variations useful for evaluating static code analysis tools, however, because the variations do not change the functionality of the test cases, we further limit our testing to just the base test case. Additionally, each test case contains a bad function containing a specific flaw and a good function that does not contain any flaws. The inclusion of both good and bad functions enables evaluation of both true positive and false positive error detection rates.

Table 4.1 shows the results of testing our software prototype using the Juliet Test Suite. The results show our monitoring approach detects all out of bounds memory accesses to dynamically allocated buffers including buffer underwrites, buffer overreads, and buffer underreads. Our prototype also detects 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. Although testing revealed our monitor to be highly effective in detecting memory errors involv-
Table 4.1: Software prototype test results using the Juliet Test Suite

<table>
<thead>
<tr>
<th>CWE</th>
<th>Description</th>
<th>False Positives</th>
<th>True Positives</th>
<th>Total Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
<td>Heap-based Buffer Overflow</td>
<td>0</td>
<td>60</td>
<td>62</td>
</tr>
<tr>
<td>124</td>
<td>Buffer Underwrite</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>126</td>
<td>Buffer Overwrite</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>127</td>
<td>Buffer Underread</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>415</td>
<td>Double Free</td>
<td>0</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>416</td>
<td>Use-after-free</td>
<td>0</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>590</td>
<td>Free Memory Not on Heap</td>
<td>0</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>761</td>
<td>Free Pointer Not at Start of Buffer</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

ing dynamically allocated regions, our approach failed to detect two memory access errors and generated one false positive result. These results are examined in the following paragraphs.

Figure 4.3 contains the relevant code from one of the two heap buffer overflow test cases our monitoring prototype did not detect. The second test case is identical except the memory copy on line 16 is alternately achieved using the `memmove` function. A `charVoid` structure containing a fixed size character array and two pointers is dynamically allocated on line 12. Our monitor successfully detected this allocation and recorded the lower and upper address range of the `charVoid` structure. The pointer initialization on line 13 results in a memory write that is successfully verified by our prototype. The undetected flaw is a memory copy on line 16 that overflows the `charFirst` buffer and partially overwrites the `voidSecond` pointer. Our monitoring approach only tracks the bounds of the `charVoid` structure and cannot detect memory errors within the structure because the monitor has no knowledge of the sub-object boundaries within the `charVoid` structure.

Our experiments with the Juliet Test Suite also revealed several limitations related to Pin, including the false positive from CWE 761 shown in Table 4.1. The false positive is generated when the monitored test case copies an environment variable into a dynamically allocated buffer. Our Pin prototype is unable to track the memory location of the environment variables because the kernel maps the environment variables into the test case’s address space before our Pin prototype begins execution. To the best of our knowledge, the current Pin framework does not support any extension that allows the address range of environment variables to be
Figure 4.3: Code snippet of the undetected heap-based buffer overflow test case

discovered at runtime. We worked around this problem by disabling ASLR and hardcoding the address range of the environment variables in our prototype.

The kernel’s program initialization sequence also prevented our software prototype from tracking the dynamic linker’s private memory segments. As explained in Chapter 2.2, the dynamic linker performs its initialization routines before the monitor prototype executes. This means the monitor cannot detect any memory allocations made by the dynamic linker because the allocations occur before our prototype begins tracking. Therefore, false memory safety violations were generated whenever the dynamic linker accessed its dynamically allocated memory segments to perform runtime address resolution operations. As a workaround, we used the Pin API to check if the target of memory access was to a region specifically designated for the dynamic linker before raising a memory violation. Although this workaround reduced the number of false positives, memory accesses to regions allocated anonymously by the linker still raised
memory violations. We note the program initialization order is a limitation of the software prototype and not the monitor design itself because the monitoring approach is designed to operate fully in parallel with the target application.

Testing with the Juliet Test Suite also yielded another interesting result involving several functions in the glibc 2.19 string library [39]. The GNU implementation of the string length, comparison, and copy functions uses an optimization that is problematic for our memory monitoring approach. Instead of operating on a single character byte at a time, the string functions are optimized to load an entire processor register with characters from memory before performing string operations. The optimization has the effect of generating false positives when the string functions are called using a dynamically allocated string whose size is not a multiple of the register’s character capacity. Highly optimized code has the potential of creating false positives and is a clear limitation of our memory monitoring approach.

Finally, testing using the Juliet Test Suite demonstrated, albeit unexpectedly, the ability of our monitoring approach to detect a class of stack-based memory errors. Stack overflows that access memory off the end of the program’s stack will not be filtered because the memory address does not fall between the stack base address and the current stack pointer. These memory accesses are subsequently checked by our monitor and a memory safety violation is raised. This result shows our monitoring approach is not just limited to detecting memory errors in dynamically allocated memory regions but is also capable of detecting stack smashing errors.

4.3 Real-World Exploit Testing

The following three subsections examine the effectiveness of our memory monitoring approach in detecting real-world attacks that exploit memory errors. The ability to detect real-world exploits is significant because it suggests that our monitoring approach is able to detect attacks against real-world systems that exploit unknown or undisclosed (also called zero-day) vulnerabilities in production software. Our results show the prototype monitor detects several real-world attacks and also supplements existing memory protections deployed in modern systems.
4.3.1 CVE-2014-0160 (Heartbleed)

The Heartbleed vulnerability is an error in the OpenSSL 1.0.1 - 1.0.1.f (inclusive) implementation of the TLS heartbeat extension [2]. The heartbeat extension is a keep-alive feature that allows both ends of a TLS connection to check if the other end is still connected and functioning properly [75]. To perform this check, a special heartbeat message is sent to the other end with an arbitrary payload. The TLS protocol specifies the other end will send back a heartbeat response message containing a copy of the payload from the received heartbeat message. The OpenSSL implementation does not verify the value in the received heartbeat message’s payload length field with the actual length of the payload before copying the payload contents into the heartbeat response message. When the payload length field of an incoming heartbeat message is larger than the actual size of the incoming payload, contents of the receiver’s memory are copied into the heartbeat response message payload. The Heartbleed vulnerability is significant because it is easy to exploit remotely, reliable, and can leak sensitive information from vulnerable targets.

To test the Pin prototype’s ability to detect the Heartbleed vulnerability, we compiled `wget` from source using OpenSSL version 1.0.1f. We also set up a malicious server that exploits the Heartbleed vulnerability whenever a client connects on port 4433. The malicious server is also configured to write the contents leaked from a vulnerable client into a log file. The software prototype was tested by monitoring the vulnerable `wget` application while it connected to the malicious server.

The results of the experiment show our prototype successfully detected exploitation of the Heartbleed vulnerability. The monitor tracked two dynamic memory allocations created for the incoming and outgoing heartbeat messages. Memory safety violations were raised when the vulnerable OpenSSL code copied memory contents from addresses beyond the bounds of the incoming message. Not only does this result show our monitor is effective in detecting real-world exploits, it also indicates the monitor could have prevented sensitive information from being leaked from applications vulnerable to the Heartbleed exploit.
4.3.2 CVE-2015-0235 (GHOST)

The `gethostbyname` family of functions are designed to perform hostname resolution and are commonly used by applications that have networking functionality. The `gethostbyname` functions take a human-readable string, such as google.com, and perform a DNS lookup to produce an IP address. However, a specially crafted hostname string can overflow an incorrectly sized internal buffer in the glibc-2.2 - glibc-2.17 (inclusive) implementation [3]. Many web servers and networked applications pass externally supplied hostname strings to the `gethostbyname` functions to verify the client, potentially allowing attackers to achieve remote code execution.

We compiled the proof of concept exploit code provided in the GHOST CVE against glibc-2.17 containing a vulnerable version of the `gethostbyname` function and used our prototype to monitor the application as it executed the exploit code. The monitor tracked the creation of the internal buffer used by `gethostbyname` and reported a safety violation when a memory write occurred past the end of the buffer. This result confirms our monitor detects buffer overwrite attacks against applications containing the GHOST vulnerability.

4.3.3 CVE-2016-4324 (LibreOffice)

The Rich Text Format (RTF) parser in LibreOffice 5.0.4 incorrectly processes files containing stylesheet and superscript tokens, potentially resulting in a use-after-free error [4]. A specially crafted RTF file will make the RTF parser dereference a value popped from an empty STL deque container. According to the CVE, the dereferenced value is controllable and can be supplied in the crafted RTF file. This allows a malicious adversary to achieve arbitrary code execution when the specially crafted RTF file is opened on the targeted system.

Using our Pin prototype, we monitored LibreOffice while opening the two proof-of-concept RTF files provided in the CVE. The first malicious RTF file did not trigger the vulnerability, but we observed consistent LibreOffice crashes when opening the second RTF file. However, the dereferenced value differed each time we opened the RTF file which suggests the value is not controllable as indicated in the CVE. In addition, the random dereference was detected
by the test system’s memory page protections, initially leading us to believe our monitoring approach did not provide any added security protection.

Further analysis revealed that our monitoring approach does provide additional security when used in conjunction with memory page protections. At the time of the crash, the LibreOffice process used 691 KB of virtual memory, 491 KB of which was dynamically allocated to LibreOffice. Assuming the value of the dereferenced pointer is controllable, there are 200 KB of virtual memory addresses accessible by an attacker that will not generate an exception due to a memory page protection violation. Because the 200 KB of virtual memory has not been explicitly allocated to LibreOffice, our monitor will raise exceptions when any of the 200 KB virtual memory addresses are accessed. This analysis illustrates our secure memory monitoring approach supplements existing memory protection techniques by providing finer grain memory access checking.

4.4 Limitations

Our memory monitoring approach shares some of the same limitations as previously proposed memory safety approaches. Dynamic memory allocations are tracked at the same granularity as previous object bounds approaches, and therefore share similar limitations. Specifically, as shown in Figure 4.3, our monitor does not have knowledge of sub-object boundaries within a dynamically allocated region of memory and cannot detect overflows or other memory errors that occur inside the tracked memory region. For example, overwriting an entry in an object’s virtual pointer table will not be detected if the virtual pointer table is located within the boundaries of an object. This limitation is common to all protection approaches that track the bounds of an object when it is dynamically allocated at runtime.

Our memory checking approach is similar to previous fat pointer techniques that only perform checks when a memory access actually occurs, such as when a pointer is dereferenced. However, unlike fat pointer techniques, our approach cannot bind a pointer to its respective memory region. As a result, our monitoring approach cannot guarantee complete spatial memory safety. In addition, memory accesses made by dereferencing dangling pointers are not
detected if the memory region to which they point is reused. This limitation is shared by nearly all previous temporal memory protection solutions.

The proposed monitoring technique also introduces several other limitations specific to our memory protection approach. Memory errors and attacks targeting filtered regions, such as the stack or data section, will not be detected. Attacks that overwrite entries in the GOT or PLT will not be detected because they reside in the program’s data section. Furthermore, memory addresses in filtered regions can be read arbitrarily, revealing potentially sensitive information about the application.

Finally, as observed during our experiments using the Juliet Test Suite, highly optimized code has the potential of generating many false positives using our memory monitoring approach. Suppressing these false positives requires removing optimizations from the source code and recompiling the application. However, this may have negative performance implications on the application and, most importantly, violates our compatibility requirement.
CHAPTER 5. HARDWARE ARCHITECTURE

This chapter 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 dynamically allocated memory regions. The profiling experiments used to design our architecture for low memory and performance overheads are also presented. The chapter concludes with a discussion of several implementation considerations for integrating our hardware monitor into a real system.

5.1 Top Level Design

![Diagram of the top level design of our hardware architecture](image.png)

Figure 5.1: Diagram of the top level design of our hardware architecture

The top level design of our monitoring hardware implementation is illustrated in Figure 5.1. 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 Figure 5.1 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 Figure 5.1 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 Allocation Tracking, Memory Access Checking, and Allocated Memory Range Storage components is discussed in the following sections.
5.2 Memory Allocation Tracking Architecture

The top level memory allocation tracking component contains several of the hardware sub-modules depicted in Figure 5.2. Figure 5.2 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 Figure 5.2 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 Figure 5.2 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.

### 5.3 Memory Checking Architecture

Figure 5.3 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,
Figure 5.3: Hardware architecture of a submodule that outputs memory addresses for checking 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, Figure 5.3 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. We also note that the filtering logic in Figure 5.3 is included within the memory checking submodule to clearly show the flow of
hardware execution. However, the filtering hardware can be implemented as one independent unit that all memory checking submodules access to reduce redundancy and minimize hardware resources.

5.4 Allocated Memory Storage Architecture

The following subsections describe 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.

5.4.1 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 by varying the number of bits [87, 93, 32, 92, 25, 30, 35, 98]. 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 monitoring prototype, our profiling tool wraps dynamic 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, and the profiling results are reported in a log file.
Table 5.1: Memory overhead of metadata bit and range pair storage formats for SPEC 2006 integer benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Max Allocs</th>
<th>Avg Size</th>
<th>Total # Bytes Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Metadata Bits</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>1,931,303</td>
<td>138</td>
<td>491,947,570</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>9</td>
<td>21,658,124</td>
<td>75,803,434</td>
</tr>
<tr>
<td>403.gcc</td>
<td>118,370</td>
<td>28,629</td>
<td>10,287,004,481</td>
</tr>
<tr>
<td>429.mcf</td>
<td>4</td>
<td>351,494,442</td>
<td>219,684,026</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>906</td>
<td>1,859</td>
<td>26,741,768</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>747</td>
<td>1,017</td>
<td>136,817,734</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>5</td>
<td>36,000,114</td>
<td>22,500,071</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>4</td>
<td>7,961,122</td>
<td>120,411,976</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>12,560</td>
<td>6,340</td>
<td>58,212,477</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>2,198,026</td>
<td>167</td>
<td>5,570,932,824</td>
</tr>
<tr>
<td>473.astar</td>
<td>313,554</td>
<td>922</td>
<td>274,144,471</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>1,904,727</td>
<td>460</td>
<td>7,779,421,581</td>
</tr>
</tbody>
</table>

Using our profiling tool, we ran the SPEC 2006 integer and floating point benchmarks and determined the total memory overheads of each storage format [42]. 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.

The results of our profiling experiments for the SPEC 2006 integer benchmarks are shown in Table 5.1. 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. Figure 5.4 provides a visual representation of the 32-bit and 64-bit range pair overheads normalized to metadata bit overheads for each 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.

Table 5.2 shows the storage requirements of each storage format using SPEC 2006 floating point benchmarks. 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. Figure 5.5 provides a visualization of the results of profiling the floating point benchmarks. Normalized values below 1.0 indicate the range pair representation has a smaller
Table 5.2: Memory overhead of metadata bit and range pair storage formats for SPEC 2006 floating point benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Max Allocs</th>
<th>Avg Size</th>
<th>Metadata Bits</th>
<th>Range Pair (32-bit)</th>
<th>Range Pair (64-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>410.bwaves</td>
<td>76</td>
<td>6,358,771</td>
<td>945,867,253</td>
<td>9,520</td>
<td>19,040</td>
</tr>
<tr>
<td>416.gamess</td>
<td>47</td>
<td>1,506</td>
<td>682,575</td>
<td>29,077</td>
<td>58,155</td>
</tr>
<tr>
<td>433.milc</td>
<td>59</td>
<td>13,560,178</td>
<td>11,039,680,016</td>
<td>52,104</td>
<td>104,208</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>104</td>
<td>235</td>
<td>16,224</td>
<td>4,424</td>
<td>8,848</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>2,056</td>
<td>725</td>
<td>2,150,277</td>
<td>189,776</td>
<td>379,552</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>5,300</td>
<td>26,507</td>
<td>433,115,684</td>
<td>1,045,728</td>
<td>2,091,456</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>69</td>
<td>19,594</td>
<td>749,823,030</td>
<td>2,449,120</td>
<td>4,898,240</td>
</tr>
<tr>
<td>444.namd</td>
<td>1,313</td>
<td>35,545</td>
<td>5,882,673</td>
<td>10,592</td>
<td>21,184</td>
</tr>
<tr>
<td>447.dealII</td>
<td>13,197,031</td>
<td>75</td>
<td>1,417,123,485</td>
<td>1,210,073,520</td>
<td>2,420,147,040</td>
</tr>
<tr>
<td>450.soplex</td>
<td>145</td>
<td>196,899</td>
<td>2,073,651,525</td>
<td>941,660</td>
<td>1,883,320</td>
</tr>
<tr>
<td>453.povray</td>
<td>11,700</td>
<td>31</td>
<td>9,291,181</td>
<td>19,322,904</td>
<td>38,645,808</td>
</tr>
<tr>
<td>454.calcuiux</td>
<td>27,148</td>
<td>5,500</td>
<td>2,645,908,048</td>
<td>30,790,040</td>
<td>61,580,080</td>
</tr>
<tr>
<td>459.gemsfdtd</td>
<td>200</td>
<td>20,474</td>
<td>1,908,710,262</td>
<td>5,966,392</td>
<td>11,932,784</td>
</tr>
<tr>
<td>465.tonto</td>
<td>14,453</td>
<td>374</td>
<td>49,459,868,588</td>
<td>8,454,949,424</td>
<td>16,909,898,848</td>
</tr>
<tr>
<td>470.lbm</td>
<td>3</td>
<td>107,200,284</td>
<td>53,600,142</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>481.wrf</td>
<td>2,409</td>
<td>313</td>
<td>20,768,374,328</td>
<td>4,244,806,536</td>
<td>8,489,613,072</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>200,682</td>
<td>1,135</td>
<td>2,018,196,969</td>
<td>113,796,464</td>
<td>227,592,928</td>
</tr>
</tbody>
</table>

Overall, our profiling experiments demonstrate the range pair storage format requires less memory overhead than storing metadata bits for 91.3% of SPEC 2006 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.

5.4.2 Range Cache Architecture

Before presenting our range caching 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 the binary search tree used in our software prototype. Hardware search trees are not especially favorable because they require longer average lookup times than caches, and they typically do

memory footprint, and storing metadata bits requires less memory overhead for benchmarks with a normalized value above 1.0.
Figure 5.5: Memory overhead of the range pair storage format normalized to metadata bit storage overhead for SPEC 2006 floating point benchmarks

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 [97, 53, 80]. Ultimately, we elected to pursue a caching architecture because caches exploit program locality that can improve overall lookup performance, and the cache control logic to support range lookup and modification is not overly complex.

Figure 5.6 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 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. Figure 5.7 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
Figure 5.7: Range cache hardware for update operations

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.

Our range cache architecture is similar to a content-addressable memory (CAM) in that every entry is searched in a fully associative manner [64]. 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 [24]. The CAMs in modern routers and switches can store hundreds of thousands of IPv4 and IPv6 routes [23]. In addition, fully-associative translation lookaside buffers (TLBs) are often implemented as CAMs, such as in the ARM Cortex-A8 [9], ARM Cortex-A9 [11], and AMD Opteron processors [51, 41]. However, TLBs are much smaller than hardware routing tables and usually contain 512 or fewer entries [65]. As shown in the following subsection, 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.

5.4.3 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 [7]. 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 [68] as a guide.

Figure 5.8 shows the hit rate results for SPEC 2006 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 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 real-world applications. The selected programs and workloads are listed in Table 5.3, and the experimental results using single level range cache configurations are shown in Figure 5.9. All workloads achieve hit rates over 90% using just a 32-entry range cache. Unlike SPEC benchmarks, increasing the range cache size steadily improves hit rates to over 98% for all real-world application workloads using a 256-entry range cache.

Experiments were also performed using multi-level range cache configurations. Specifically, we analyzed the behavior of an exclusive two-level range cache whose L2 capacity is twice as large as the L1 size. Due to its exclusive caching property, the overall hit rate plots of the two-level range cache follows the same trajectory as the single-level hit rate plots and are omitted to avoid redundancy. Figure 5.10 illustrates the hit rates of L2 range cache accesses for real-world workloads. As the L2 range cache size is increased, the hit rate decreases until its size reaches 128 entries, at which point the hit rates begin to increase dramatically. It is not clear what is causing this trend, however, we believe the replacement algorithm and the L2 cache size relative to the L1 size may play a crucial role in the behavior shown in the figure. Further experiments
Table 5.3: Real-world applications and workloads

<table>
<thead>
<tr>
<th>Application</th>
<th>Workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox</td>
<td>Visit google.com</td>
</tr>
<tr>
<td>Evince</td>
<td>Scroll through a PDF</td>
</tr>
<tr>
<td>SSH Client</td>
<td>Execute <code>ls -l</code> on remote</td>
</tr>
<tr>
<td>VLC Media Player</td>
<td>30 seconds of MP3 playback</td>
</tr>
<tr>
<td>LibreOffice Calc</td>
<td>View a spreadsheet</td>
</tr>
<tr>
<td>LibreOffice Writer</td>
<td>View a word document</td>
</tr>
<tr>
<td>LibreOffice Impress</td>
<td>View a presentation</td>
</tr>
</tbody>
</table>

Figure 5.9: Single level range cache hit rates for real-world applications

and analysis are needed to determine a suitable L2 cache size and replacement algorithm to obtain higher hit rates.

We also attempted to estimate 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 [9, 18, 65]. Assuming a clock speed of 2.0 GHz, we find the total runtime overhead of a 256-entry range cache is only 2.9\% 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.

5.5 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. 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 [43]. 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 kernel 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.

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.
CHAPTER 6. RELATED WORK

Protection mechanisms can be broadly classified into three categories: probabilistic approaches, integrity mechanisms, and memory safety. In the following sections, we discuss both hardware and software implementations of each protection category. We also provide a discussion of metadata caches related to our architecture.

6.1 Probabilistic Approaches

Probabilistic approaches provide protection using randomization and secrets. Instruction Set Randomization encrypts executables with a unique key at compile time, and the processor decrypts the executable at runtime [50]. Encrypting the executable disguises the processor’s instruction set architecture which makes it difficult for an attacker to develop proper machine code for a code-injection attack. Thekkath et al. develop a processor that encrypts all data with a unique key before it is transferred to memory and decrypts the data when it re-enters the processor [85]. To maintain performance, processors that encrypt data or instructions use relatively simple encryption algorithms ([50] XORs instructions with the key and [85] uses DES encryption.) As a result, the keys can be quickly discovered using brute-force techniques as demonstrated by [81].

Similar encryption techniques have also been proposed in software. PointGuard is a compiler technique that encrypts all pointers in memory and decrypts them when they are used by the processor [28]. Bhatkar and Sekar extend PointGuard by encrypting all program data in memory to mitigate information leaks [13]. However, encryption at the software level has not been widely adopted because it is not source code or binary compatible.
Address Space Layout Randomization (ASLR) is the most widely deployed probabilistic protection technique. ASLR is a software technique that randomizes the location of an executable’s code and data areas in memory [66, 21]. This makes control flow attacks less reliable because the address of the attack payload is always at different (random) locations. To decrease the entropy, attackers will place multiple copies of attack payload in the virtual address space using heap-spraying [94] or JIT-spraying [14] techniques, or exploit a format string vulnerability to leak information about the memory layout [62]. Brute-force or de-randomization attacks have been shown to be effective against 32-bit systems [79]. Additionally, distribution specific ASLR implementations can be exploited to reduce entropy. For example, most Linux distributions only randomize the location of shared libraries, not the executable itself. The base address of the executable can be randomized if the program is compiled as a Position Independent Executable (PIE), but will incur a 10% performance penalty on average [67]. Unlike previous probabilistic techniques, our monitoring approach does not rely on randomization or secrets to protect dynamically allocated memory regions.

6.2 Integrity Mechanisms

The goal of an integrity mechanism is to prevent a program’s execution from being hijacked by an attacker. The general hijacking approach is to exploit a stack-based buffer overflow to overwrite a control variable on the stack (such as the return address) to gain control of the program’s execution [63]. Early hijacking attacks executed attack payload directly on the stack. However, modern processors no longer allow stack memory to be executed. To bypass this restriction, attackers reuse code from libraries or the program itself.

Various code reuse methods have been proposed and used in real-world exploits. Return-to-libc attacks reuse sections of C library code that make system calls using attacker controlled arguments [95]. Return-oriented programming (ROP) is driven by the execution of return instructions and chains together small sections of a program’s code (called gadgets) to carry out attacks [78]. Unlike return-to-libc attacks, ROP has been proven to be Turing complete [88, 73]. A generalized version of ROP that is driven by indirect calls is called Jump-oriented programming (JOP) and was first introduced in [19, 15].
Several software-based approaches to prevent program hijacking have been proposed. Stack canaries are a compile-time approach that place special values between stack allocated buffers and return addresses. Overflowing a stack allocated buffer will change the canary value, which is checked before a return instruction is executed. StackGuard [27] and ProPolice [37] use stack canaries to detect stack-based buffer overflows that lead to code reuse attacks. Although stack canaries have extremely low runtime overhead (less than 1%), they are vulnerable to information leaks, direct overwrite, and do not provide protection against JOP attacks that utilize indirect calls [84].

Shadow stacks are another software approach that address the pitfalls of stack canaries. Return addresses are copied to a separate shadow stack in memory and are compared with the normal stack’s return addresses when the program executes a return instruction. Any discrepancy between the two copies of the return address indicates that a buffer overflow has occurred [91, 22]. Shadow stacks do not necessarily prevent hijacking attacks; they simply make it harder to carry out an attack because the attacker has to corrupt the return address in two locations.

Abadi et al. introduce the Control-Flow Integrity property that, when enforced, is a comprehensive solution to hijacking attacks [5]. All valid return sites in an application are statically determined and stored inside the program using binary rewriting techniques. At runtime, all program return addresses are checked against the predetermined, valid return locations at 15-45% overhead. Control-Flow Integrity enforcement is not binary compatible and presents significant challenges with dynamic libraries and position independent applications.

Several specialized hardware architectures have been proposed to defend against hijacking attacks. DIFT [83] marks all external program input as tainted as well as any program data that is manipulated by tainted data. DIFT prevents program hijacking by prohibiting the processor from executing memory that has been marked as tainted. DIFT has been accelerated in [48, 20] and generalized to support custom taint tracking policies in [92, 29]. SCRAP uses signature-based detection to detect hijacking attacks. A simple state machine and hardware return address stack are used to detect ROP and JOP attacks [49].
To thwart early hijacking attacks that execute code directly on the stack, modern processors allow memory regions to be marked as non-executable using a hardware no execute (NX) bit. Data Execution Prevention (DEP) uses the NX bit to prevent execution of program data memory regions and was first deployed in Windows XP Service Pack 2 [8]. Write-XOR-Execute, the Linux equivalent to DEP, also uses the NX bit to ensure that a memory page is either writable or executable, but never both [90]. Code reuse attacks are able to bypass protections enabled by the NX bit because reused code exists in executable memory regions. Additionally, programs written in languages that perform just-in-time compilation cannot be protected by the NX bit. The host machine code that is generated at runtime must be written to a memory location that can also be executed by the processor. Our work secures accesses to dynamically allocated memory regions and cannot prevent an application’s execution from being hijacked because control variables are located on the stack, a region that our monitor excludes from memory access checks.

6.3 Memory Safety

Memory safety focuses on detection and prevention of two types of memory errors. A spatial memory error occurs when a program accesses memory outside of its designated memory space. The classic example of a spatial memory error is a buffer overflow, where a program reads or writes data beyond the boundaries of a buffer. A temporal memory error occurs when a program references an object that has been previously deallocated. Dereferencing a pointer to an object that has already been freed, or a use-after-free vulnerability, is an example of a temporal memory error. Systems that implement both spatial and temporal safety are said to have complete memory safety, which is ideal for software protection.

6.3.1 Spatial Safety

Spatial safety is achieved by forcing a pointer to stay within the bounds of the object to which it points. There are four bounds checking techniques: tripwires, coloring, fat pointers, and object bounds.
Tripwires are special bytes that surround allocated objects in memory. When a pointer that points to a tripwire is dereferenced, an exception occurs to signal the spatial violation. The Purify compiler tool inserts tripwires into existing code at an average cost of 2.3x runtime overhead [40]. SafeMem utilizes Error-Correction Code memory to create tripwires, reducing the performance cost to at most 15% [69]. MemTracker makes use of a custom hardware mechanism to further reduce overheads to less than 5% [93]. 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 tag pointers and their corresponding memory regions with unique identifiers. When a program calls a memory allocation function, the memory and pointer returned by the allocation function are assigned the same, unique color. During a dereference, the pointer’s color must be the same as the color of the memory to which it points. A hardware assisted coloring implementation is described in [25] with under 10% performance cost. Similar hardware designs with comparable overheads are presented in [30, 35]. The 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 two different memory allocations will share the same color. Spatial violations involving a pointer that moves between two memory regions with the same color will remain undetected.

Fat pointers [60, 46, 57, 55, 32, 38, 96, 56, 71, 52] are the most widely explored approach for providing spatial memory protection. Fat pointers extend pointer representation to include the lower address 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. 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.

Jim et al. propose a new programming language, Cyclone, that incorporates the use of fat pointers to ensure spatial memory safety [46]. CCured [60] and Milewicz et al. retrofit existing source code with fat pointer representation and runtime checking of pointers [55]. SoftBound
stores additional pointer metadata in a disjoint memory space and is the most efficient software implementation of fat pointers to date, having 67% runtime and 64% memory overheads [57].

Hardware implementations of fat pointers aim to further reduce the performance and memory costs. HardBound [32], SafeProc [38], and Watchdog [56] propose extensions to existing hardware to support fat pointer representation. HardBound expands the size of the register file and utilizes a special cache and TLB to support fat pointers, while SafeProc and Watchdog use micro-operations to query a hardware bounds table. Intel Memory Protection Extensions (MPX) provide fat pointer support starting with the Skylake microarchitecture [71].

New hardware architectures that support fat pointers have also been proposed. Kwon et al. propose a new ISA that uses floating point representation to support fat pointers. Although their technique is efficient, it limits the number of valid pointers in the system [52]. The Cheri capability system also supports bounds information for pointers [96].

To address the compatibility challenges of fat pointers, object bounds trade a small amount of spatial safety for compatibility. When an object is created by the program, the upper and lower memory bounds of the object are recorded. Object bounds techniques maintain compatibility by storing bounds information in a disjoint memory space. To ensure pointers point to their intended object, all pointer arithmetic operations are checked against the bounds information. Note that the pointer does not have to be dereferenced to cause a spatial memory violation.

One problem with the object bounds approach is that valid pointer arithmetic will generate false positives. For example, when using a pointer to iterate through an array, the last loop iteration will typically move the pointer past the end of the array. Even though no spatial memory violation occurs (because the pointer is not dereferenced), object bounds checking will report a violation. Jones and Kelly address this problem by modifying the GCC compiler to pad tracked objects with an extra byte [47]. A generalized solution to the problem points out of bounds pointers to a special object [74, 34].

Baggy Bounds Checking [6] and PAriCheck [99] trade memory for performance by forcing object sizes to be powers of two. This allows for more compact bounds storage and efficient lookup. Currently, these are the two most efficient software implementations of object bounds
with runtime overheads of 60% and 49%, respectively. Additionally, Yong et al. enforce object 
bounds using dynamic binary instrumentation, ultimately demonstrating that object bounds 
tracking is also binary compatible [98].

AHEMS implements object bounds tracking in hardware using an asynchronous coprocessor 
arquitecture [89]. Runtime overhead is only 10% 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.

The main benefit of object bounds is compatibility with unsupported or third-party code 
and libraries. However, the approach does not provide full spatial protection because memory 
inside objects (such as sub-objects, arrays, or structures) cannot be protected.

6.3.2 Temporal Safety

Temporal safety is enforced by detecting dangling pointers and use of uninitialized memory. 
A dangling pointer is a pointer that continues to point to an object after it has been freed in 
memory. Dereferencing a dangling pointer is called a use-after-free vulnerability and can cause 
unpredictable code bugs and memory corruption. Worse yet, use-after-free vulnerabilities are 
frequently exploited in zero-day attacks. Uninitialized memory errors occur when a program 
reads from memory before it has been initialized. Reading uninitialized memory can cause 
undefined behavior as well as leak data from the previous object that was allocated at the 
address.

Purify [40] is one of the first tools to provide temporal memory safety. Its compiler based ap-
proach averages 2.3x runtime overhead. Memcheck [77] is the first widely used temporal safety 
tool and is built on the Valgrind dynamic binary instrumentation framework [61]. Memcheck 
does not require the target program to be recompiled, but suffers 20-30x runtime overhead. 
Dr. Memory [17] has similar functionality to Memcheck but is based on the multithreaded binary 
instrumentation platform DynamoRIO [16]. Due to the efficiency of DynamoRIO, Dr. Memory 
is twice as fast as Memcheck. AddressSanitizer detects use-after-free vulnerabilities at just 73% 
runtime cost but requires recompilation of source code [76]. MemorySanitizer uses static comp-
iler instrumentation to check for use of uninitialized memory, and its LLVM implementation
has 2.5x runtime overhead. The authors state that using MemorySanitizer with AddressSanitizer provides the same functionality as Memcheck and Dr. Memory at lower runtime cost, but requires source code to be recompiled [82].

Unfortunately, these approaches do not detect all use-after-free vulnerabilities because they associate temporal information with memory regions instead of pointers. If the memory pointed to by a dangling pointer gets reused, the tools remark the memory as valid and will not detect the temporal error when the dangling pointer is dereferenced. CETS solves this problem by associating temporal information with pointers at 48% runtime cost [58]. However, CETS requires spatial safety to also be enforced, so the total runtime overhead will be much greater. Our monitoring approach provides temporal safety as well as spatial safety using object bounds, and unlike all previous work, our technique does not require the monitored application to be recompiled or instrumented.

### 6.4 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. [93] investigate three types of metadata cache configurations. Split caching stores metadata in a separate, dedicated cache. Shared caching places regular program data and metadata in the same cache, and interleaved caching adds metadata bits to the corresponding cache line in the data cache. Performance of the three cache configurations is analyzed using difference cache and metadata sizes. The authors find a 2 KB two-way set-associative split cache to provide the best balance of performance and area cost [93]. Devietti et al. also deploy a 2 KB split metadata cache with a dedicated metadata TLB [32].

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 [92]. The same metadata cache configurations are used in [30, 35].
Several unconventional cache architectures have also been proposed. Similar to classical data caches, the previously mentioned metadata cache architectures use addresses to lookup cached metadata tags. Harmoni uses instruction opcodes to lookup cached metadata tags [31]. Dhawan et al. implement metadata tag compression to better utilize cache space [33].

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 [87]. Our caching architecture stores metadata in a range pair format similar to Tiwari et al., however, our hardware does not perform range coalescing and does not require code recompilation or binary instrumentation to capture and lookup heap allocation metadata.
CHAPTER 7. CONCLUSION

Memory errors in computer software are serious threats to security. Applications built using languages that permit manual memory management, such as C and C++, are prone to exploitable memory errors that enable attackers to compromise a system. Despite the protections integrated into today’s computing systems, real-world attacks continue to emerge that exploit memory errors and bypass all currently deployed defenses.

Many runtime checking approaches have been proposed to detect memory access errors in C and C++ software applications. Software-based solutions impose large performance and memory overheads that often offset the security protection provided by the technique. Hardware-based solutions offer better runtime performance but require code to be recompiled in order to utilize the custom architecture. Source code recompilation is necessary for hardware and software approaches and is an unrealistic requirement for legacy software or applications that use proprietary code, libraries, or plug-ins. The performance and compatibility limitations of existing solutions have limited their adoption in practice.

This thesis introduced a 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 showed how to leverage the processor’s calling convention to track calls to dynamic memory allocation routines and validate heap memory accesses. A hardware implementation for tracking memory allocations was presented, and a range cache architecture for filtering and verifying memory accesses was introduced. A software prototype was constructed using the Pin framework, and we evaluated the security effectiveness of our architecture using the NSA Juliet Test Suite and real-world exploits. The prototype was also used to select a low overhead allocation storage format and to measure the hit rates for various sizes of our range cache architecture.
Our experience with the software prototype shows it is able to track eight different classes of heap memory errors and two real-world attacks that exploit memory errors in dynamically allocated memory regions. Our monitoring approach also provides enhanced protection when used alongside currently deployed defenses, and can also detect overflows that run off the end of the stack. However, experiments revealed that the approach has similar limitations as previous work that provides memory safety at object bounds granularity. Errors that occur within sub-objects cannot be detected, and full spatial safety cannot be achieved because pointers are not bound to a specific memory region. Despite these limitations, our technique detected 98% of heap memory errors in testing, indicating it is still effective in practice.

Memory overhead experiments demonstrated that storing pairs of addresses corresponding to dynamically allocated regions has a smaller memory footprint than metadata bits for 91.3% of SPEC benchmarks. Simulation experiments of our range cache architecture using SPEC benchmarks and real-world workloads produced hit rates of over 95% for a 256-entry range cache, and the additional runtime overhead was only 2.9%. However, further investigation of multi-level range cache configurations is needed to improve L2 hit rates, including varying the L2 range cache size relative to the L1 size and exploring different cache replacement policies.

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 multiple processes can also be added to the monitoring hardware as well as uninitialized memory checking functionality 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.
BIBLIOGRAPHY


[95] R. N. Wojtczuk. The advanced return-into-lib(c) exploits: PaX case study. Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e, 2001.


