Preventing Zero-Day Exploits of Memory Vulnerabilities with Guard Lines

Sterling Vinson
The Johns Hopkins University
Applied Physics Laboratory
sterling.vinson@jhuapl.edu

Rachel Stonehirsch
The Johns Hopkins University
Applied Physics Laboratory
rachel.stonehirsch@jhuapl.edu

Joel Coffman[*]
The Johns Hopkins University
Applied Physics Laboratory
joel.coffman@jhu.edu

Jim Stevens[†]
University of Maryland
Department of Computer Science
jims@umd.edu

ABSTRACT

Exploitable memory errors are pervasive due to the widespread use of unsafe programming languages, such as C and C++. Despite much research, techniques for detecting memory errors at runtime have seen limited adoption due to high performance overhead, incomplete memory safety, or non-trivial microarchitectural changes.

This paper describes Guard Lines, a hardware / software memory error detector that detects common types of spatial and temporal memory errors at runtime without imposing a significant performance penalty (on average only 4%). Guard Lines provides memory safety by defining certain regions of memory as inaccessible "guards," which are created in software during memory allocation. If a program ever accesses guarded memory, the hardware raises an exception indicating a memory safety violation. Guard Lines requires minimal microarchitectural changes, and it uses a novel metadata design to efficiently track the guard locations. This paper describes the design, implementation, security analysis, and performance evaluation of Guard Lines and demonstrates its feasibility to protect real-world applications against exploitable memory vulnerabilities.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Hardware security implementation**; • **Software and its engineering** → *Software testing and debugging*; • **Computer systems organization** → Architectures.

[*]Work completed at the Johns Hopkins University Applied Physics Laboratory. Joel Coffman is currently with the Department of Computer and Cyber Sciences at the United States Air Force Academy.
[†]Work completed at the Johns Hopkins University Applied Physics Laboratory. Jim Stevens is currently with the University of Maryland, College Park.

KEYWORDS

Guard Lines, memory safety, hardware support, buffer overflows, AddressSanitizer

ACM Reference Format:
Sterling Vinson, Rachel Stonehirsch, Joel Coffman, and Jim Stevens. 2019. Preventing Zero-Day Exploits of Memory Vulnerabilities with Guard Lines. In *Software Security, Protection, and Reverse Engineering Workshop (SSPREW9), December 9–10, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3371307.3371311

1 INTRODUCTION

Memory errors are pervasive due to the ongoing use of unsafe languages, such as C and C++, in the implementation of operating systems (OSes), applications, and even language runtimes. C and its immediate descendants emphasize a uniform approach to handling strings and arrays in terms of pointers, which complicates efficient optimizations due to the difficulty of determining the objects being referenced [26]. More importantly, C places the burden of memory management on the programmer, with no explicit support for bounds-checking pointer references and supporting dynamically-allocated storage only by library routines.

Despite these issues, C remains one of the most popular programming languages due to its limited abstractions of hardware and high performance compared to other languages, as well as the enormous investment in legacy C-based software. Consequently, memory errors remain prevalent with buffer overflows perennially topping the lists of most dangerous software errors.[1] Lack of memory safety is estimated to cost billions of dollars annually in damages and lost revenue. Eliminating memory errors would have an enormous impact on cybersecurity because stack and heap errors account for only 50% of reported vulnerabilities but 90% of exploited vulnerabilities [36]. Moreover, large-scale malware infections are often based on exploit toolkits where 63% of exploits stem from memory errors [36].

Given the importance of memory safety, academic research and commercial products have sought to address this issue, but most have seen limited success: they raise the difficulty of attacks for a short period, but adversaries quickly adopt more sophisticated techniques, which commonly use memory corruption as the initial

[1]CWE/SANS Top 25 Most Dangerous Software Errors: https://www.sans.org/top25-software-errors/

path to compromise. Defensive techniques are hamstrung by two issues: a requirement to support legacy code bases written in unsafe languages and an unwillingness to accept decreased performance in exchange for improved security. The adoption of defensive techniques often dictates a performance impact less than 5–10% [34]. Consequently, this paper introduces Guard Lines, a novel approach to prevent common types of memory errors with low performance overhead and support for legacy applications.

Guard Lines detects memory errors through the use of "guards," which are placed around allocated objects and inside deallocated objects. The CPU checks these guards on each memory access, raising an exception to the OS if the accessed region of memory is guarded. Using Guard Lines only requires programs to be recompiled to insert instructions to add guards for statically-allocated variables and stack frames; linking against a runtime library for dynamically-allocated memory protects variables on the heap. The Guard Lines design detects linear memory corruption and use-after-free errors with minimal space and time overhead.

The major contributions of Guard Lines are as follows:

- Guard Lines does not require code changes to existing software. Protection for statically-allocated variables and stack frames only requires recompilation. Heap protection works with existing dynamically-linked binaries.
- Guard Lines uses minor hardware extensions to minimize its performance overhead. Guard Lines's metadata design allows metadata to be checked in parallel with every memory access, which is essential for good performance.
- Guard Lines protects against real-world memory vulnerabilities such as Heartbleed.
- Preliminary results indicate that Guard Lines's performance impact is acceptable—approximately 4% for the benchmarks used in our evaluation.

The design of Guard Lines is flexible, supporting the implementation of eXecute only Memory (XOM) [19] and hiding code pointers without requiring trampolines [10, 11] or pointer mangling [24]. In conjunction with code diversification, these protections can prevent code reuse attacks, including just-in-time return-oriented programming (ROP) (JIT-ROP), without directly modifying code pointers.

The remainder of this paper is organized as follows. Section 2 provides background material and reviews existing commercial attempts to address memory safety. Section 3 describes the design and implementation of Guard Lines. We provide a detailed security analysis of Guard Lines in Section 4. Section 5 provides an evaluation of the Guard Lines concept. Section 6 discusses related work, focusing on similar approaches and academic research. Finally, we conclude in Section 7.

2 BACKGROUND

The security implications of memory corruption were recognized no later than 1972 [1]. Nevertheless, hardware at the time was too slow to enforce memory safety, particularly at the granularity of individual objects, so memory safety in C was left to the programmer, and memory safety violations proliferated. Since the Morris Worm in 1988, memory errors, such as buffer overflow and use-after-free vulnerabilities, have been the root cause of countless cyberattacks.

Memory safety is the restriction of memory accesses to prevent memory corruption due to software faults, which are defects or flaws in a software component. Memory safety provides the following guarantees [4, 15, 40]:

- a program never reads uninitialized memory,
- a program does not invoke any illegal heap operations (e.g., invalid or double frees),
- a program does not access freed memory (e.g., through dangling pointers), and
- a program does not dereference pointers to addresses not associated with the referenced variable (e.g., buffer overflows).

A memory error is any access to an object using a pointer expression different from the one that is intended [35]. There are two types of memory errors: spatial violations, which are accesses outside the bounds of the intended object, and temporal violations, which are accesses to an unallocated or deallocated object.

Software developers and hardware vendors have attempted to address memory errors using a variety of techniques, including new programming languages, middleware, and hardware technologies. While some solutions offer perfect memory safety and others make corruption—and subsequent exploitation—more difficult, no solutions satisfy the requirements for both perfect memory safety and high performance for legacy code bases. The remainder of this section describes these existing solutions in more detail.

2.1 Programming Languages

Programming languages like Ada and Java enforce memory safety with bounds checks on all memory accesses and automatic memory management. Other languages like Rust do not allow null pointers or dangling pointers in safe code; the compiler ensures that all accesses to such memory are safe. None of these languages has replaced unsafe languages, such as C and C++, because their memory safety features typically decrease performance, the language itself may be unsuited to low-level systems programming, and vast amounts of legacy code are already written in unsafe languages.

2.2 Software Technologies

Stack and heap canaries are a runtime technique to prevent buffer overflows. A canary [9] is an unpredictable, dynamically-generated value inserted at the end of a buffer. At key points during execution, such as when returning from a function or using a heap-allocated object, the canary value is checked against a shadow copy. If the canary has changed, then the application is considered corrupted. Canaries have limited value because they can be defeated with information disclosure vulnerabilities, they provide no protection against read-based buffer overflows, and they are often disabled because of their performance overhead.

Other software solutions to mitigate the effects of memory corruption include Microsoft's Enhanced Mitigation Experience Toolkit (EMET) and Control Flow Guard. EMET uses dynamic binary instrumentation (DBI) to add checks on critical function calls (e.g., VirtualProtect) to detect conditions that are similar to ROP attacks.[2] Control Flow Guard is a compile-time instrumentation technique that adds checks to indirect calls to constrain the attacker to valid transitions in a program's control flow graph as defined by the compiler. Unfortunately, both techniques can be subverted by attackers through memory errors (e.g., Control Flow Bending [7]).

Tools like Purify [14] and Valgrind Memcheck [29] use DBI to monitor all memory accesses and detect memory corruptions. These tools suffer from high performance overhead (increasing runtime by factors of 5-110 and 9–130 respectively [22]) due to the frequency of instructions that access memory.

2.3 Hardware Changes

Restrictions on executable memory (e.g., the No-eXecute bit (NX bit) and Write XOR Execute (W^X)), make it more difficult for attackers to inject executable code directly into a vulnerable program. Nevertheless, attackers can bypass these protections using code reuse attacks (e.g., ROP [30] and related techniques [6–8, 13, 27]) to call a memory protection function in the OS. Ultimately, as long as attackers can corrupt memory, they can cleverly manipulate the system and bypass defenses.

The most successful memory safety enforcement in hardware is the guard page, which provides extremely coarse-grained buffer overflow protection by marking virtual pages as "not present" in between certain data structures. Linux places a guard page between each process's stack and heap, and the OpenBSD memory allocator places a guard page after all allocations larger than 2 kilobytes. Guard pages are unsuitable for protecting smaller objects because they waste enormous amounts of physical memory due to padding.

Fine-grained memory corruption detection and prevention is even less widely used. The original x86 architecture had a BOUND instruction that checked the upper and lower bounds of a pointer, but it was not widely used by C compilers due to high overhead, and it was removed from the x86-64 specification. Intel re-introduced bounded pointers with the Memory Protection Extensions (MPX), but the performance overhead averages 50% [23], and this capability is not currently used by any major software applications.

Silicon Secured Memory (SSM) in the SPARC architecture uses memory tagging to associate tag numbers with every pointer and cache line. The hardware verifies that the two tags match for every memory access. SSM detects common spatial and temporal safety memory errors with low performance overhead, but SPARC's small market share limits its impact.

3 GUARD LINES

Guard Lines detects memory errors by marking certain memory locations as inaccessible "guards." Guards are placed outside of allocated objects and inside deallocated objects, so common types of memory errors (e.g., linear buffer overflow and use-after-free errors) always read or write to a guard. The hardware immediately detects any access to a guard location and raises an exception, stopping the program before it can be exploited. Guard Lines supports testing to detect memory access faults prior to the release of software as well as operational use in deployed systems to prevent memory exploits, including zero-day attacks.

[2]A ROP attack [30] constructs sequences of instructions, or "gadgets," from the program's code and executes these gadgets to turn the program's original code into an interpreter capable of arbitrary computation.

```
// Allocate a 10-byte buffer.
char * buffer = malloc(10);
// Get a string from user input.
char * name = promptUserForName();
// Copy input string into buffer.
strcpy(buffer, name);
// Release the memory used by the buffer.
free(buffer);
```

Figure 1: An example of unsafe code. If the input string (i.e., the user's name) exceeds the size of the buffer, then it overwrites adjacent memory.

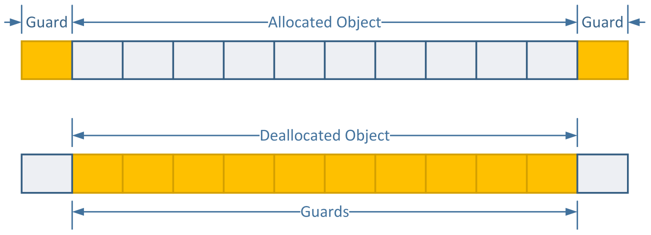


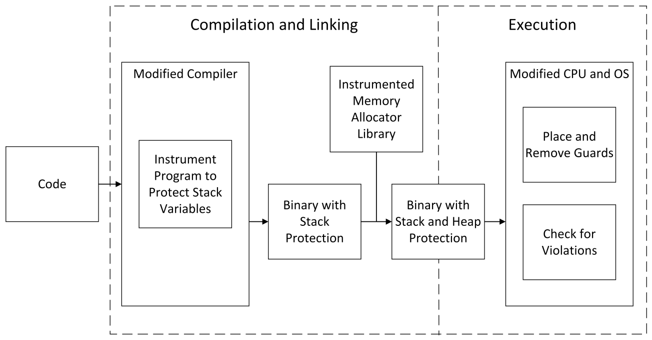
Figure 2: Example guard locations. Allocation places guards before and after the object to prevent overflow. Deallocation guards the entire object to prevent use-after-free.

The example program in Figure 1 contains a potential memory error: if the user-provided string is longer than 10 characters, then the copy to buffer overflows. Guard Lines inserts a guard at the end of the buffer as shown in Figure 2, so when the overflow attempts to write to the guard, an exception is raised and the program is halted.

Guard Lines is simple to use with existing software. Figure 3 depicts the workflow for Guard Lines which involves (1) recompiling the software to protect data on the stack, (2) linking to a custom memory allocation library to protect data on the heap, and (3) executing the recompiled software on a system with hardware and OS support for Guard Lines. We briefly describe these steps in the following paragraphs.

Building on our prior example, the sample program (Figure 1) is recompiled by a compiler with Guard Lines support. The compiler inserts instructions to create and remove guards during program execution to protect statically-allocated variables (e.g., name), including global variables, and stack frames. The compiled program links to a runtime library that uses guards to protect dynamically-allocated memory (e.g., buffer) on the heap. Using both the compiler instrumentation and the runtime library provides the best protection, using just one only provides partial protection, and using neither provides no protection beyond that offered by the original platform.

In addition to compiler support, Guard Lines requires support from both hardware and the OS. Guard Lines extends the instruction set architecture (ISA) to add the custom instructions that create and remove guards, and it extends the virtual memory system to add the metadata that defines guards. The CPU checks the guards



Compilation and Linking
Execution
Code
Modified Compiler
Instrument Program to Protect Stack Variables
Instrumented Memory Allocator Library
Binary with Stack Protection
Binary with Stack and Heap Protection
Modified CPU and OS
Place and Remove Guards
Check for Violations

Figure 3: Guard Lines workflow. Source code is recompiled for stack protection, and an instrumented memory allocator provides heap protection. The CPU checks for memory errors, and the OS is responsible for handling runtime exceptions.

in parallel on each memory access, and if an accessed region is guarded, the CPU raises an exception to the OS, which handles the exception by terminating the program or recovering from the memory error.

Guard Lines systems are compatible with non-Guard Lines software, allowing continued use of legacy applications. Even if the CPU and OS support Guard Lines, guards will not be set when running software that has not been compiled to use these protection mechanisms; the program executes normally, even if it contains memory safety violations.

The remainder of this section provides further details about the design of Guard Lines, describing the changes required to the CPU, OS, and compiler in turn.

3.1 CPU

Hardware changes to the CPU center around the metadata required to track guards and the new instructions to manipulate that metadata. We focus on the design of the metadata because it is critical to Guard Lines's efficiency.

3.1.1 Metadata. Guard Lines requires metadata to track inaccessible regions of memory (i.e., guards). Performance dictates that accessing this metadata causes minimal extra memory reads, which are a major cause of poor performance in other shadow memory schemes. This is accomplished by placing all metadata in the same cache lines as regular data.

A two-level metadata structure defines the guard locations. The first level, located in the page table, indicates which cache lines in a page contain guard metadata. The second level, located in the cache line, indicates which bytes in the cache line are inaccessible. The following paragraphs describe the design of both levels in more detail.

The top-level metadata is the guard line mask (GLM), which is added to every page table entry (PTE) as shown in Figure 4. Each GLM is associated with a single page, and each bit in the GLM corresponds to a single cache line in that page. A set bit in the GLM indicates that the cache line contains additional metadata

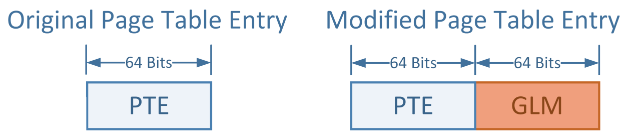


Figure 4: Guard Lines modifications to a page table entry on a 64-bit architecture. The guard line mask (GLM) doubles the size of the page table entry (PTE) from 8 bytes to 16 bytes.

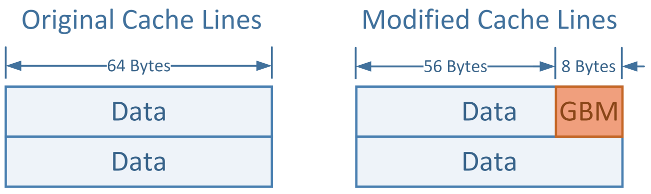


Figure 5: If a cache line contains guards, then the last 8 bytes of the cache line are used for the guard byte mask (GBM).

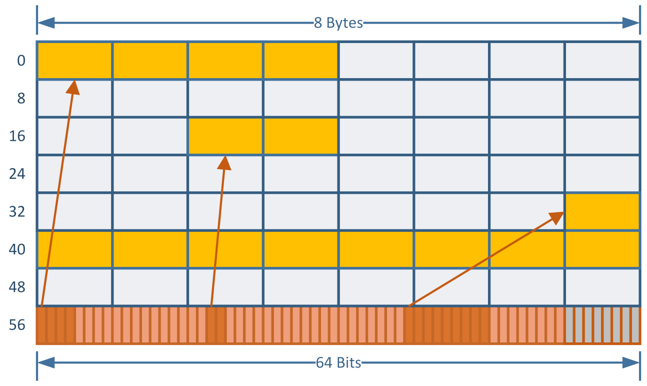


Figure 6: A partially guarded cache line shown in a two-dimensional representation. The last 8 bytes are the guard byte mask (GBM). Bits set in the GBM define guard locations in the first 56 bytes. The last 8 bits of the GBM are reserved.

specifying the guarded bytes in that line. For a standard-sized 4096-byte page containing 64 64-byte cache lines, the GLM is 64 bits.[3] When there is a page fault, the PTE and GLM are loaded together from the same cache line and cached together in the translation lookaside buffer (TLB). In a multi-level page table, only the GLM in the last table is used, and GLMs in higher level PTEs are ignored if present.

When accessing virtual memory, the memory management unit (MMU) retrieves both the PTE and the GLM from the TLB or page table. It uses the PTE for address translation, and it checks the GLM to see if the accessed cache line contains guards.

When a bit is set in the GLM, the corresponding cache line contains additional metadata called the guard byte mask (GBM),

[3]The design of Guard Lines is compatible with larger page sizes (e.g., 2 MB pages), but size of the GLM must increase to accommodate the larger page size. Guarded objects may reside in any size page.

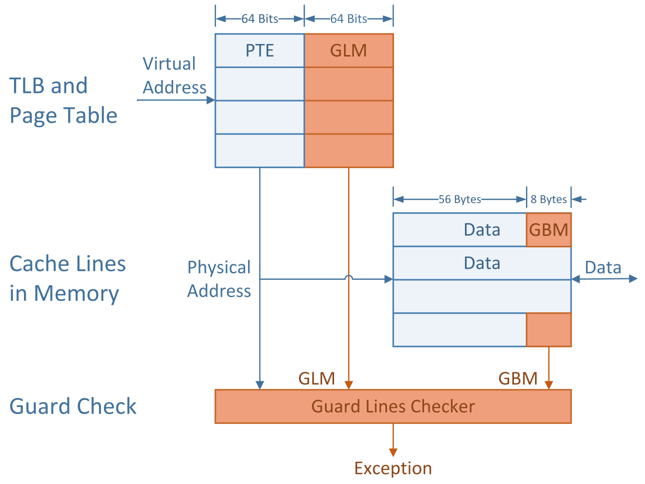


Figure 7: Hardware Modifications.

shown in Figures 5 and 6. The GBM is 64 bits and located at the end of the 64-byte cache line. The first 56 bits of the GBM correspond to the first 56 bytes in the cache line. If any of the GBM bits are set, then the corresponding bytes are guard bytes. The CPU raises an exception if a guard byte or the GBM is ever accessed by a load or store instruction. The last 8 bits of the GBM are currently ignored and reserved for future extensions of Guard Lines.

3.1.2 Instruction Set. Guard Lines requires new instructions to modify the protected metadata in the GLM and GBM. The following instructions efficiently set and clear bits in the metadata to create and remove guards:

`GB_OR addr, mask` Creates guard bytes in a cache line. Each set bit in the mask corresponds to a guarded byte. If the cache line is not already a guard line, it sets the bit in the GLM and initializes the GBM with the mask value. Otherwise, it `OR`s the GBM with the mask.

`GB_NAND addr, mask` Removes guard bytes in a cache line. Each set bit in the mask corresponds to a byte that should no longer be guarded. It `NAND`s the GBM with the mask.

`GB_QUERY addr, dest` Checks for guard bytes in a cache line and places a bitmask of the guard locations into a register.

`GL_NAND addr, mask` Removes all guards from select cache lines in a page. Each set bit in the mask corresponds to a cache line that should no longer be guarded. It `NAND`s the GLM with the mask.

These instructions only modify the GLM and GBM. Guarding or unguarding a byte does not affect that byte's contents. The unguarding instructions (`GB_NAND` and `GL_NAND`) raise an exception if they are used at a location that is not guarded.

The new instructions are straightforward to implement. Reading or writing the GBM is equivalent to a regular load or store because the GBM address is easily calculated from the operand address. GLM writes must be implemented in the TLB and page table walker (PTW) because the GLM is part of the PTE.

3.1.3 Other Changes. Figure 7 shows how Guard Lines modifies the virtual memory and cache hardware to store metadata and enforce guards.

Placing the GLM in the PTE increases the PTE size, which requires modifications to the PTW and the TLB. The PTE size is doubled, so the page table size is also doubled, while the number of PTEs in each page table stays the same. Every PTE load also caches the GLM for that page in an expanded TLB.

The cache checks the guard metadata in parallel with each load or store operation. If any of the accessed bytes are guarded (i.e., if the corresponding metadata bits in the GLM and GBM are set), then the CPU raises an exception. If a violation occurs as a result of a speculatively executed instruction (e.g., in a Spectre attack [17]), then the CPU must prevent the memory access without raising an exception.

3.2 OS

The OS is responsible for creating and managing page tables, so it must increase the PTE size to include the new GLM field. The OS also handles Guard Lines exceptions raised by the CPU by terminating the program or recovering from the memory error.

Runtime Allocator. The runtime memory allocator (e.g., the C standard library) uses guards to protect dynamically allocated memory. Allocation functions place boundary guards before and after each allocation to prevent overflow. Deallocation functions remove the boundary guards then guard the entire deallocated region to prevent use-after-free. The allocator eventually removes these guards when it reallocates the memory. Once the guards are removed, use-after-free errors are undetectable, so reallocation should be delayed as much as possible. In the sample program (Figure 1), `buffer` is dynamically allocated with `malloc` and deallocated with `free`, which place guards as shown in Figure 2.

It is also possible to guard memory before it is first allocated to prevent uninitialized use errors. However, this is not practical because completely guarding large memory regions requires one write for each cache line to set the GBMs.

3.3 Compiler

The compiler protects globals and stack variables by surrounding them with guards. It adjusts the memory layout so there is at least one unused byte between adjacent variables and so there is room for the GBM in cache lines that contain guards.

The compiler inserts instructions in the program to manage these guards at runtime. The program creates guards around global variables at start-up and removes them when the program terminates. Guards on the stack are created and removed when calling and returning from a function. In the sample program (Figure 1), the pointers to `buffer` and `name` are stack variables, so the compiler guards both of them.

Alternatively, the compiler can use "guard canaries" to protect entire stack frames rather than individual objects. Guard canaries serve the same purpose as conventional stack canaries [9] but have several advantages. Regular stack canaries can only detect if the canary value is modified, they do not alert until the canary is checked, and they can even be bypassed if an attacker knows their value and correctly overwrites them. Guard canaries, however, alert immediately if they are read or overwritten. Guard canaries are easier to implement than protection for individual objects and have better performance in some cases, but they do not detect overflow between objects in the same stack frame.

4 SECURITY ANALYSIS

Guard Lines defends against exploitable memory vulnerabilities. We assume that the attacker is aware of these vulnerabilities and is able to exploit them in unprotected software (e.g., by providing malicious input to the program). This section defines a formal adversarial model, describes how Guard Lines prevents memory errors, and identifies limitations of this approach.

Adversarial Model Following prior work [27, 30, 33], we define an attack as having two steps:

(1) exploit a vulnerability to subvert the program's normal control flow and
(2) execute arbitrary computation to cause the program to accomplish the attacker's purpose.

Memory safety is primarily concerned with the first stage of the attack—that is, altering the program's intended control flow. To do so, attacks typically rely on the ability to write arbitrarily to memory at least once during the program's execution [7, 33].

Clearly the absence of memory errors or, more generally, perfect memory safety precludes such attacks. As stated in Section 2, perfect memory safety provides the following guarantees [4, 15, 40]:

- a program never reads uninitialized memory,
- a program does not invoke any illegal heap operations (e.g., invalid or double frees),
- a program does not make any accesses to freed memory (e.g., through dangling pointers), and
- a program does not dereference pointers to addresses not associated with the referenced variable (e.g., buffer overflows).

Guard Lines approximates perfect memory safety by addressing the last 2 guarantees. It prevents linear spatial access violations and temporal violations to previously-freed memory. We describe both in more detail in the following paragraphs and provide a real-world case study of Guard Lines's effectiveness in Section 5.

Programs with full Guard Lines protection have guards at the boundaries of all local, global, and dynamically-allocated variables. Recently deallocated regions are also guarded, and guard canaries protect stack frame boundaries.

A buffer overflow vulnerability, like in Figure 1, allows an attacker to write past the end of a buffer and overwrite sensitive data such as return pointers to hijack the program. Guard Lines prevents buffer overflow exploits because it is impossible to write linearly past a buffer boundary without accessing a guard. When the overflow tries to access the byte immediately before or after the buffer, Guard Lines detects the violation and prevents the attack.

A dangling pointer vulnerability, a type of use-after-free vulnerability, allows an attacker to modify a freed object that the program continues to use. For example, if a program frees a dynamically-allocated object but later calls one of its methods, the attacker can modify the freed object to overwrite its function table and hijack the program. Guard Lines prevents dangling pointer use by guarding recently freed objects. Once an object is guarded, the program can no longer call its methods, and the attacker cannot overwrite its function table.

4.1 Limitations

Guard Lines does not detect non-linear buffer overflows to valid locations outside a guarded buffer because these accesses skip over the guards placed around buffers. This is particularly problematic for globals and stack variables, which are usually allocated close together in a predictable order with small guards between them. One possible mitigation is to use random allocation (e.g., DieHard [4]) for all buffer objects to make successful non-linear overflows unlikely. Another option is to add software bounds checks wherever Guard Lines alone is insufficient. The compiler could perform either of these mitigations automatically.

In addition, Guard Lines does not detect use-after-free exploitations if the memory has been reallocated. Reallocating previously freed memory clears all the guards in the new allocation, so pointers that still exist from previous allocations in the same region can be reused. This issue can be mitigated with randomized allocation and less heap reuse. For example, the runtime allocator may not reallocate memory until after a number of additional allocations [14]; coupled with a randomized allocation strategy, it would be very difficult for an adversary to reliably overwrite an existing object's function pointers to redirect control flow.

Finally, Guard Lines does not detect intra-object overflow. While it is possible to place guards inside an object (a `struct` in C), prior experience indicates that such protections must be suppressible because some C programs assume such data is contiguous [14].

4.2 Metadata Protection

The Guard Lines metadata must be protected, otherwise an attacker could potentially clear every guard before causing a memory corruption. GBMs are always guarded, so accessing one without using a Guard Lines instruction raises an exception. Although the GLMs are not guarded, they are inaccessible to user programs because they are located in the page table, which is protected by the OS. The OS's normal memory protection mechanisms protect a program's metadata from being accessed by other programs.

Any user program can use Guard Lines instructions to modify its own guard metadata, so an attacker that gains control of a program can unguard memory to set up the next stage of the attack. The unguarding instructions make this more difficult by raising an exception if the attacker tries to remove guards that do not exist. Nevertheless, Guard Lines is primarily concerned with stopping the first stage of an attack, when attackers are unable to arbitrarily execute unguarding instructions.

5 EVALUATION

We evaluate Guard Lines by (1) testing its ability to detect memory errors in a case study of the Heartbleed bug and (2) assessing its performance overhead using the PARSEC benchmark suite.

5.1 Implementations

We implemented Guard Lines in the x86-64 and RISC-V [39] architectures. We modified Quick Emulator (QEMU) [3] to emulate a

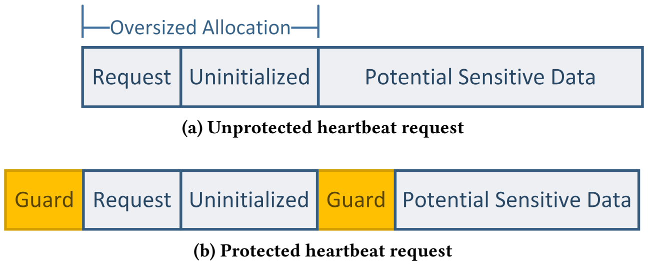


Figure 8: (a) The request is placed inside of an oversized buffer. Data outside of the buffer can be read. (b) Guards are placed around the oversized buffer. Data outside of the buffer is protected.

CPU with Guard Lines extensions for x86-64. A custom runtime library wraps the libc allocation functions to provide heap protection, while a modified Clang / Low Level Virtual Machine (LLVM) [18] compiler uses guard canaries to protect stack frames. For RISC-V, we prototyped Guard Lines in actual hardware using the Rocket CPU core [2]. We doubled the page table size in the RISC-V Linux kernel to support the GLM and modified the Newlib runtime library to provide heap protection. We did not add stack protection to the RISC-V compiler.

These Guard Lines implementations were straightforward and did not require extensive changes, as expected from the lightweight design. For example, the changes to the page table in RISC-V Linux only affect 15 lines of code.

5.2 Case Study: Heartbleed

The Heartbleed bug was a serious vulnerability in OpenSSL's transport layer security (TLS) heartbeat implementation that enabled attackers to remotely read protected server memory like user passwords and secret keys [38]. In this case study, we demonstrate how Guard Lines protections can defeat Heartbleed attacks.

The heartbeat protocol allows machines to remain in contact with each other by sending "heartbeat" messages. Clients send heartbeat requests containing a payload message and a message length value, and servers respond with the same message. If the provided message length is larger than the actual message, the request is invalid, and the server should not respond.

Servers vulnerable to Heartbleed copy the request message into the response without verifying that the provided message length matches the actual message size. A Heartbleed attack uses an invalid request to cause a buffer overflow which copies potentially sensitive data into the response.

Guard Lines prevents the most serious Heartbleed attacks, but it does not prevent every overflow because of OpenSSL's unusual memory management strategies. The compiler and runtime library automatically guard every allocation, but OpenSSL uses a custom internal memory allocator that places some objects in oversized allocations. As a result, some Heartbleed attacks effectively only cause intra-object overflows, which are not detected. In this case, the attack may leak data from other users' previous requests, but it cannot leak more sensitive data like private keys.

When an OpenSSL server receives a heartbeat request, it places the request in an oversized 16-kilobyte buffer, as shown in Figure 8 (a). Next, the server copies the request message into the response. If the provided message length is larger than the actual message length, then it overflows the request message and begins reading the data that follows it in memory, with a maximum read size of 64 kilobytes.

A successful Heartbleed attack must overflow the larger request buffer in order to leak sensitive data in memory. Guard Lines prevents this overflow by placing guards around the oversized buffer as shown in Figure 8 (b). The overflow accesses the guard, triggering an exception that stops the server before it sends the response, so no sensitive data is released.

Programs that allocate every object individually get full Guard Lines protections with no code changes, while programs that violate this rule, such as OpenSSL, only receive partial protections.[4] OpenSSL could eliminate the effective intra-object overflow by using allocations of the correct size or placing guards inside of the oversized allocations.

We demonstrated this case study on both the RISC-V CPU and the x86-64 emulator and showed that Guard Lines defeats Heartbleed attacks.

5.3 Benchmarks

To assess the software performance overhead of Guard Lines, we used the PARSEC 3.0 benchmark suite [5]. A modified Clang / LLVM compiler inserts guard canaries to protect stack frames, but does not protect individual local variables. A custom runtime library wraps the libc allocation functions using `LD_PRELOAD` to guard every allocation and every deallocated region in the heap.

We did not benchmark either of the full Guard Lines implementations because the x86-64 QEMU version is not reflective of actual hardware, and the RISC-V version has limited compatibility with available benchmark suites. Instead, we used stock x86-64 hardware, with all custom Guard Lines instructions in the compiler and runtime library replaced with store instructions. We expect that the performance overhead due to hardware is small relative to the software overhead, so these benchmarks roughly approximate the overall performance overhead for Guard Lines. These assumptions are discussed further in Section 5.4.

The benchmarks were all compiled with Clang version 3.6.2 and run with the PARSEC native input set. We did not use facesim, ferret, freqmine, raytrace, and x264 in our evaluation: facesim and raytrace would not compile using Clang 3.6.2, and AddressSanitizer did not work correctly with ferret, freqmine, and x264, which precluded comparison using these programs. To evaluate the performance of Guard Lines we compared it against the following configurations:

Native Default settings with no additional protections
Stack Canaries Stack canaries only
Guard Lines Canaries Guard canaries only
Guard Lines Heap Guard Lines heap protection only
Guard Lines Full Guard canaries and heap protection

[4]Naturally, the applicability of Guard Lines heap protection depends upon the number of programs that use standard allocation functions (e.g., `malloc` and `free`) instead of providing their own replacements in a custom memory allocator. Our assumption that most programs use standard allocation functions is consistent with other research in this field (e.g., [4]).

Table 1: Mean performance overhead for each configuration

Configuration	Overhead (%)
Stack Canaries	0.1
Guard Lines Canaries	0.8
Guard Lines Heap	3.2
Guard Lines Full	4.1
AddressSanitizer [28]	35.8

AddressSanitizer AddressSanitizer [28] only
Figure 9 shows the benchmark results, and Table 1 lists the average (weighted arithmetic mean [16]) overhead for each configuration.

Guard Lines Full has an average overhead of 4.1%. Most of this is attributable to heap protection (3.2%), while guard canaries add negligible overhead (0.8%). Guard Lines performs significantly better than AddressSanitizer (averaging 35.8% overhead), with lower overhead for every benchmark program. Stack canaries, on the other hand, has the best performance with only a 0.1% overhead. However, they offer the weakest security guarantees and are easily defeated. Notably, Guard Lines satisfies the performance requirement for practical security techniques, an overhead of less than 5–10% [34].

Programs that make more allocations have higher overhead for both Guard Lines and AddressSanitizer. canneal has the highest overhead for Guard Lines (22%) because it makes millions of very small allocations. This is close to a worst-case for Guard Lines because of the high number of guard instructions and the amount of memory reserved for guards. Programs that make fewer allocations, such as blackscholes and dedup, have negligible overhead for Guard Lines.

5.4 Discussion

Guard Lines minimizes overhead for metadata checks by placing all metadata in the same cache lines as regular data. The GLM is in the same cache line as the PTE, and the GBM is in the same cache line as the guards that it defines. This extreme spatial locality means that the hardware can access metadata with no additional memory reads and check for guards in parallel with every memory access.

The added Guard Lines instructions introduce some overhead, although this overhead is mitigated by the high spatial and temporal locality with other memory accesses. One guard instruction is required for each cache line when creating guards. Each heap allocation requires just 1 or 2 guard instructions, but deallocations require 1 guard instruction per cache line to set guards in the entire freed region.

The added metadata increases memory usage and disperses data, which can reduce performance. The page table size is doubled to include the GLM, and the GBM requires 8 bytes at the end of the cache line when it is present. GBMs could be present in every cache line, so the maximum memory usage for GBMs is one-eighth of a program's memory. The actual memory usage depends on the allocation pattern. There is 1 or 2 guarded cache lines per heap allocation, which results in an overhead of one-eighth of memory for small allocations and negligible overhead for large allocations.

The guard bytes themselves can also increase memory usage. At least one unused byte must be present before and after each object. In many cases, those unused bytes are already present because of alignment padding, so there is no memory overhead. When not enough padding is present, it is necessary to adjust the memory layout to make room for guards or GBMs.

6 RELATED WORK

Memory errors have been a serious security issue for decades, and many solutions have been proposed to improve memory safety. Section 2 reviews several existing commercial techniques. The following section discusses academic efforts that address this issue.

6.1 AddressSanitizer

AddressSanitizer [28] is the primary inspiration behind Guard Lines. Both use inaccessible guards in memory (called "poisoned redzones" in AddressSanitizer) to detect spatial and temporal memory errors. Unlike Guard Lines, AddressSanitizer is implemented entirely in software, using compile-time instrumentation (CTI) to create redzones and check them on each memory access. It stores its metadata in a contiguous shadow memory region which maps to the entire memory space. The instrumentation required to monitor memory accesses is on the order of 5–10 instructions at minimum, and the metadata checks require additional memory accesses to load the metadata from shadow memory. As a result, AddressSanitizer suffers from significant performance overhead, with an average slowdown of 73% and a 3x increase in memory usage [28], making it more suitable for testing and debugging than for deployment.

Guard Lines, on the other hand, is designed to avoid these causes of overhead. Metadata checks are performed by the hardware, and the guard metadata is located near the guards to avoid extra memory reads. In addition, Guard Lines is more granular and does not require as much memory for metadata—only an eighth of some cache lines instead of an eighth of the entire virtual memory space. With just 4% average overhead, Guard Lines could be used in deployed systems. Furthermore, AddressSanitizer can only monitor memory accesses in binaries compiled with AddressSanitizer instrumentation enabled. Guard Lines offers full heap protection for existing dynamically-linked binaries, with additional stack protection available for recompiled code.

6.2 REST

Random Embedded Secret Tokens (REST) [32], like Guard Lines, is a hardware technique used to implement AddressSanitizer's memory error detection algorithm with lower performance overhead. Both REST and Guard Lines avoid shadow memory and maximize spatial locality to reduce overhead for guard checks. Rather than external metadata, REST uses the memory contents to determine if a memory region is blacklisted. If a cache line's contents match a secret token value, then that cache line is a token, and any memory access triggers an exception.

REST has been more extensively evaluated than Guard Lines, and it currently has better performance (2% overhead compared to 4%), but Guard Lines offers several advantages. REST is less granular, with 64-byte tokens (16 bytes in an alternate design) compared to 1 byte in Guard Lines. This wastes significant memory for the tokens

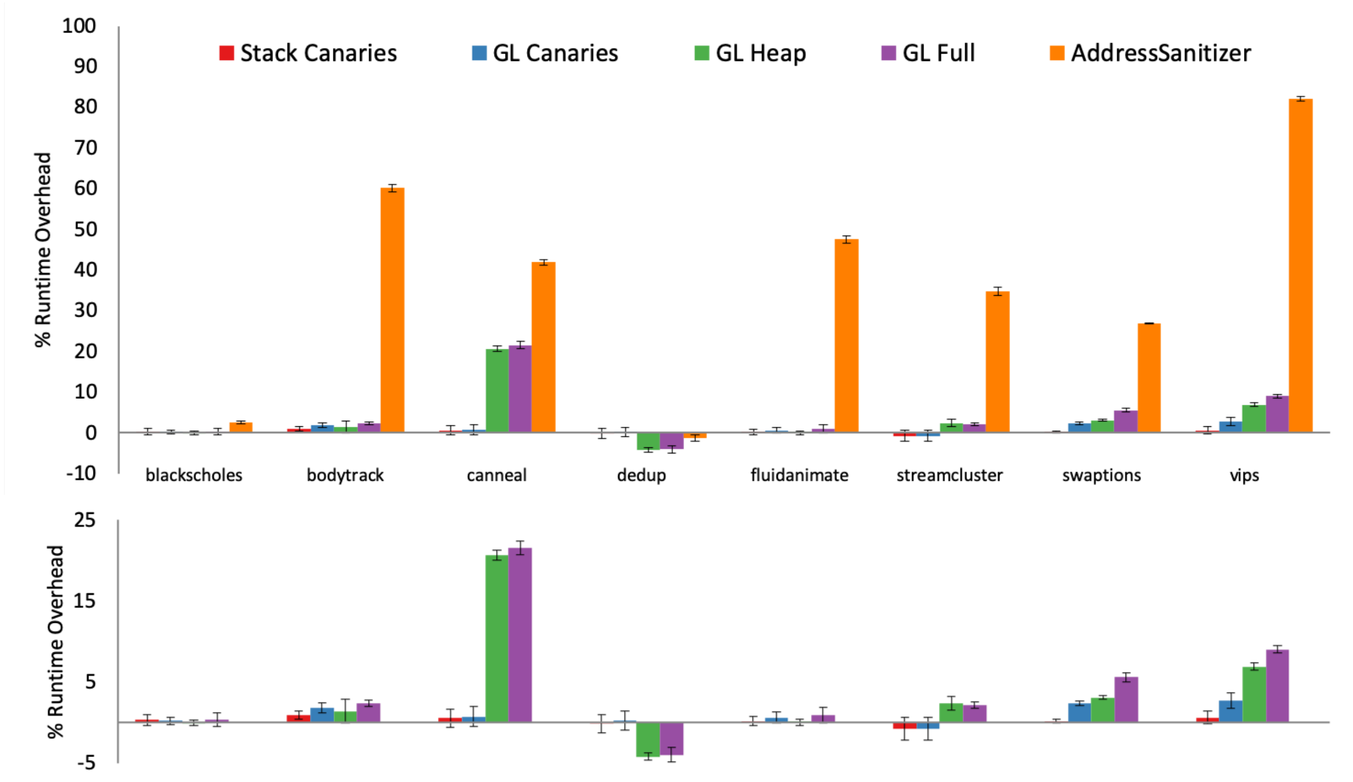


Figure 9: Mean performance overhead for 20 runs of each benchmark (lower is better). Error bars indicate the standard deviation of the runs. The bottom graph omits AddressSanitizer to highlight the overhead of the various Guard Lines configurations.

themselves and for padding each allocation to the end of the cache line. This padding can cause false negatives in REST that do not occur in Guard Lines. There is also a small chance of false positives in REST if memory contents in a program happen to match the random secret token value. More critically, the design of Guard Lines allows extensions to support additional use cases, such as XOM.

6.3 Pointer Checking

Pointer checking is a class of techniques in which metadata, such as an upper and lower bound, is associated with each pointer. Three prominent hardware-based pointer checking technologies are Hardbound [12], Watchdog [20], and Intel MPX [23]. Watchdog provides complete spatial and temporal memory safety, while Hardbound and MPX just provide complete spatial protection. All three use disjoint metadata stored in a shadow memory space.

Their performance varies: MPX has 50% overhead on average [23], Watchdog has 24% overhead [20], and Hardbound has just 5% overhead [12]. A major cause of this overhead is the use of disjoint metadata. Additional loads are required to retrieve metadata from the shadow space. Shadow memory is also inefficient in terms of storage because the entire address space is shadowed even if only a small percentage of the address space stores pointers. This contributes to a high memory overhead, as shown by Watchdog [20], which has a 56% memory overhead. These approaches also require extensive microarchitectural changes, i.e. adding caches for the pointer metadata and injecting micro-ops to check and manage the metadata. WatchdogLite [21], a variant of Watchdog, requires minimal hardware changes, but as a result has more software complexity and slightly worse performance than Watchdog (29% overhead compared to 24%).

6.4 Others

MemTracker [37] uses a programmable state transition table to monitor accesses to memory addresses and has on average a 2.7% performance overhead. However, it requires significant hardware changes including new pipeline stages and a new cache for the state metadata.

HeapMon [31] is a similar approach that uses a helper thread to keep track of memory state and monitor for memory errors. It detects heap memory bugs with good performance (5% average overhead), but it also requires substantial hardware support (cache extensions and communication queues).

SafeMem [25] uses error-correcting code (ECC) metadata in DRAM to detect illegal memory accesses. No hardware changes are required, but it only works at cache line granularity, and enabling or disabling protection is fairly slow because it requires a write to main memory.

7 CONCLUSION

This paper describes Guard Lines, a hardware / software memory error detector that eliminates common types of memory errors with acceptable performance overhead (on average only 4%). Guard Lines combines the hardware mechanism and performance of guard pages with the fine-grained redzone algorithm used by AddressSanitizer. The novel design of the guard metadata maximizes the locality of reference to avoid extra memory accesses for guard checks. Guard Lines does not require extensive microarchitectural changes, and it provides heap protection for legacy binaries without recompilation.

Guard Lines prevents many types of memory errors, but it does not offer complete memory safety by itself. However, it is possible to combine it with other technologies to improve its security guarantees. For example, adding layout randomization or software bounds checks approximates complete spatial memory safety. Though we do not explore such extensions here, the design of Guard Lines is also sufficiently flexible to support XOM and hiding code pointers, which, in combination with other techniques such as code diversification, can prevent code reuse attacks.

Guard Lines can also be used as a general-purpose memory tagging technique. The existing design can easily be extended to support new policies by changing the meaning of the lower-level metadata (the GBM). Potential applications include bounded pointers providing perfect memory safety, thread synchronization enforcement, multi-level security, and hints to the CPU to improve prefetching and cache utilization.

With its simple design and low overhead, Guard Lines is viable for use in production environments, where it would significantly improve computer security. The combination of hardware and software changes mean that widespread adoption would be a lengthy process. Nevertheless, we believe that the ongoing risk posed by memory errors warrants this investment.

REFERENCES

[1] James P. Anderson. 1972. *Computer Security Technology Planning Study Volume II.* Technical Report ESD-TR-73-51.

[2] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track.* USENIX Association, Berkeley, CA, USA, 41–46. https://www.usenix.org/legacy/events/usenix05/tech/freenix/bellard.html

[4] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06).* ACM, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[5] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[6] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11).* ACM, New York, NY, USA, 30–40. https://doi.org/10.1145/1966913.1966919

[7] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15).* USENIX Association, Berkeley, CA, USA, 161–176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10).* ACM, New York, NY, USA, 559–572. https://doi.org/10.1145/1866307.1866370

[9] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium.* USENIX Association, Berkeley, CA, USA, 63–78. https://www.usenix.org/legacy/publications/library/proceedings/sec98/cowan.html

[10] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15).* IEEE Computer Society, Washington, DC, USA, 763–780. https://doi.org/10.1109/SP.2015.52

[11] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15).* ACM, New York, NY, USA, 243–255. https://doi.org/10.1145/2810103.2813682

[12] Joe Devietti, Colin Blundell, Milo M.K. Martin, and Steve Zdancewic. 2008. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08).* ACM, New York, NY, USA, 103–114. https://doi.org/10.1145/1346281.1346295

[13] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15).* ACM, New York, NY, USA, 901–913. https://doi.org/10.1145/2810103.2813646

[14] Reed Hastings and Bob Joyce. 1992. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 USENIX Conference.*

[15] Michael Hicks. 2014. What is memory safety? Online: http://www.pl-enthusiast.net/2014/07/21/memory-safety/ (Accessed: 06 March 2018). http://www.pl-enthusiast.net/2014/07/21/memory-safety/

[16] Lizy Kurian John. 2004. More on Finding a Single Number to Indicate Overall Performance of a Benchmark Suite. *SIGARCH Computer Architecture News* 32, 1 (March 2004), 3–8. https://doi.org/10.1145/991124.991126

[17] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (January 2018), 16. arXiv:1801.01203

[18] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO '04).* IEEE Computer Society, Washington, DC, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[19] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. *SIGPLAN Notices* 35, 11 (November 2000), 168–177. https://doi.org/10.1145/356989.357005

[20] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12).* IEEE Computer Society, Washington, DC, USA, 189–200.

[21] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14).* ACM, New York, NY, USA, 175:175–175:184. https://doi.org/10.1145/2581122.2544147

[22] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Transactions on Programming Language Systems* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[23] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *arXiv.org* abs/1702.00719 (June 2017), 24. http://arxiv.org/abs/1702.00719

[24] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kRˆX: Comprehensive Kernel Protection Against Just-In-Time Code Reuse. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17).* ACM, New York, NY, USA, 420–436. https://doi.org/10.1145/3064176.3064216

[25] Feng Qin, Shan Lu, and Yuanyuan Zhou. 2005. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *HPCA '05.* IEEE Computer Society, Washington, DC, USA, 291–302. https://doi.org/10.1109/HPCA.2005.29

[26] Dennis M. Ritchie. 1993. The Development of the C Language. In *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. ACM, New York, NY, USA, 201–208. https://doi.org/10.1145/154766.155580

[27] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 745–762. https://doi.org/10.1109/SP.2015.51

[28] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. USENIX Association, Berkeley, CA, USA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[29] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 17–30. http://static.usenix.org/events/usenix05/tech/general/seward.html

[30] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[31] Rithin Shetty, Mazen Kharbutli, Yan Solihin, and Milos Prvulovic. 2006. HeapMon: A Helper-thread Approach to Programmable, Automatic, and Low-overhead Memory Bug Detection. *IBM J. Res. Dev.* 50, 2/3 (March 2006), 261–275. https://doi.org/10.1147/rd.502.0261

[32] Kanad Sinha and Simha Sethumadhavan. 2018. Practical Memory Safety with REST. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, 600–611. https://doi.org/10.1109/ISCA.2018.00056

[33] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 574–588.

[34] László Szekeres, Mathais Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[35] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2011. *Memory Errors: The Past, the Present, and the Future.* Technical Report IR-CS-73.

[36] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '12)*, Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova (Eds.). Springer, Berlin, Heidelberg, 86–106. https://doi.org/10.1007/978-3-642-33338-5_5

[37] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. 2007. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 273–284. https://doi.org/10.1109/HPCA.2007.346205

[38] J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu. 2015. Risk Assessment of Buffer "Heartbleed" Over-Read Vulnerabilities. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 555–562. https://doi.org/10.1109/DSN.2015.59

[39] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[40] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709