MemGuard: Securing Memory in the Presence of Memory Vulnerabilities

David Armanious
Brown University

Vasileios P. Kemerlis
Brown University

Abstract

Recent studies have shown that information hiding even in large virtual address spaces can be bypassed. Inspired by software fault isolation, MemGuard provides a security guarantee regarding the integrity and secrecy of a single memory region of predefined size by instrumenting untrusted memory loads and stores. On Linux x86-64, MemGuard introduces an average overhead of 5.2% and code size increase of 7.8% for software-based instrumentation. We show how MemGuard can be used as a building block for even stronger security guarantees to harden an application.

1 Introduction

Memory vulnerabilities have been plaguing programs written in memory-unsafe languages such as C and C++ for years. This can be disastrous when either the secrecy of the data is required or when the integrity of the data is required, as is often the case when layering security mechanisms. For example, a major assumption of the stack canary defense is that the master canary is both not known to the attacker and cannot be controlled by the attacker [6]. Similarly, if an attacker can control the memory where a shadow stack is stored, she has successfully bypassed that defense mechanism [3].

The majority of approaches to maintain the security of a safe region fall into two categories: those that are protected deterministically via some form of software fault isolation (SFI) [17, 15, 4, 14], and those that are protected probabilistically by hiding within the entropy of the virtual address space [11, 12]. In this battle between information hiding and SFI, the former has been gaining popularity, as can be seen by the push toward more fine-grained randomization schemes and even re-randomization at runtime [10, 7, 18].

Unfortunately, the security based on information hiding has been shown repeatedly to be much weaker than expected by a number of exploits and side channels attacks [13, 8]. Most troubling is the presence of fundamental hardware-based side-channel attacks that disclose the address of isolated regions of memory [9], completely undermining the very idea of information hiding.

MemGuard implements a practical SFI model to secure a region of memory in a user process with the single address space requirement of having sufficient memory at the lowest accessible portion. Adapting and improving on kRˆX's approach for implementing read or execute in the kernel [14], MemGuard cleverly instruments all unsafe memory reads and writes to achieve a low average overhead of just above 5%.

Section 2 describes the adversarial model. We describe the design of MemGuard in Section 3, including requirements of the safe region, the instrumentation performed, how violations can be handled, and the various optimizations applied. We then evaluate MemGuard's performance in Section 4 and discuss its security and uses in Section 6. We briefly survey select related works in Section 7 and conclude in Section 8.

2 Threat Model

Attacker Capabilities

- The attacker cannot compromise the program at load time
- The attacker cannot compromise the kernel
- The attacker is not using a side-channel attack
- The attacker has access to an arbitrary memory read and an arbitrary memory write that can be exploited at any point during the program's execution

Target Assumptions

- The target's application enforces $W \oplus X$ to prevent code injection
- The target's kernel is a trusted code base
- The target's methods marked safe contain no memory vulnerabilities

3 Design

3.1 Safe Region Properties

In the general case, we must ensure for every untrusted memory read or write that the memory address is not within the safe region. The straightforward solution is to compare the memory address against both the upper and lower bounds of the safe region. If the address falls within the safe region, we have detected a violation and handle it appropriately.

One requirement of the safe region is that it begins at precisely `vm.mmap_min_addr`. Due to an increase in kernel null pointer dereferences, ret2usr attacks gained popularity; in response, kernels began designating a fixed size of memory at the bottom of the address space as inaccessible [16]. In Linux, this is known as `vm.mmap_min_addr`, accessible via the `sysctl` system administration command, and has a typical default value of `0x10000` (16 KiB). Any call to `mmap` involving pages from this region will fail, and any attempt to reference memory in that region will similarly fail.

By requiring the safe region to be at the lowest mappable portion of the address space, we immediately halve the instrumentation by eliminating the need to check the lower bound. Any address below the upper bound will either lie within the safe region or below the `vm.mmap_min_addr`, both of which constitute a violation that should trigger a fault.

The second major requirement of the safe region is its size: for reasons detailed in Section 3.4.1, there must be 2 GiB of guard pages at the upper bound of the safe region (or simply 2 GiB of unmapped memory above the safe region). Any access to those 2 GiB will trigger a fault.

3.2 Instrumentation

For any instruction that can read from or write to memory that is not within a method explicitly designated to be safe, we compare the memory address being accessed to the upper bound using a `cmp` instruction and issue a "fault" if a violation occurs. This occurs in three steps:

1. **Load address.** Because `cmp` cannot calculate the effective address in the comparison, we must first load the effective address (`lea`) into a reserved scratch register, which we chose to be `%r15`. A memory operand for x86 consists of the following: `%SegReg:$Disp(%BaseReg, %IdxReg, Scale)`. By design, MemGuard cannot instrument memory accesses using a non-default segment, as this violates the simple linear view of the virtual address space the instrumentation assumes. We execute a `lea $Disp(%BaseReg, %IdxReg, Scale),%r15` instruction where each component of the memory address is known at compile-time.
2. **Compare against upper bound.** We simply execute a `cmp $upper_bound,%r15` instruction.
3. **Conditional jump.** We execute a `jge` instruction with the target being the instruction we are instrumenting. If `%r15` is below `upper_bound`, an access violation has occurred, the jump is not taken, and the violation handler code is executed. In either case, unless the violation handler terminates the program or thread, execution will continue normally.

Of course, maintaining the semantics of the program requires spilling any registers that can be modified by the instrumentation code. In particular, the `%rflags` register must be preserved for every instrumentation code path; and the `%rdi` register must be preserved in the case of access violations. This is accomplished by simple executing `pushf`, `push %rdi`, `popf`, and `pop %rdi` instructions where appropriate.

Finally, we must note that it is possible to conditionally read from memory using the `cmov` instruction. In these cases, we rewrite the `cmov` into a conditional jump followed by the equivalent `mov` instruction, as we must only check the address if the memory access will actually occur.

3.3 Handling Violations

MemGuard supports four violation handling behaviors:

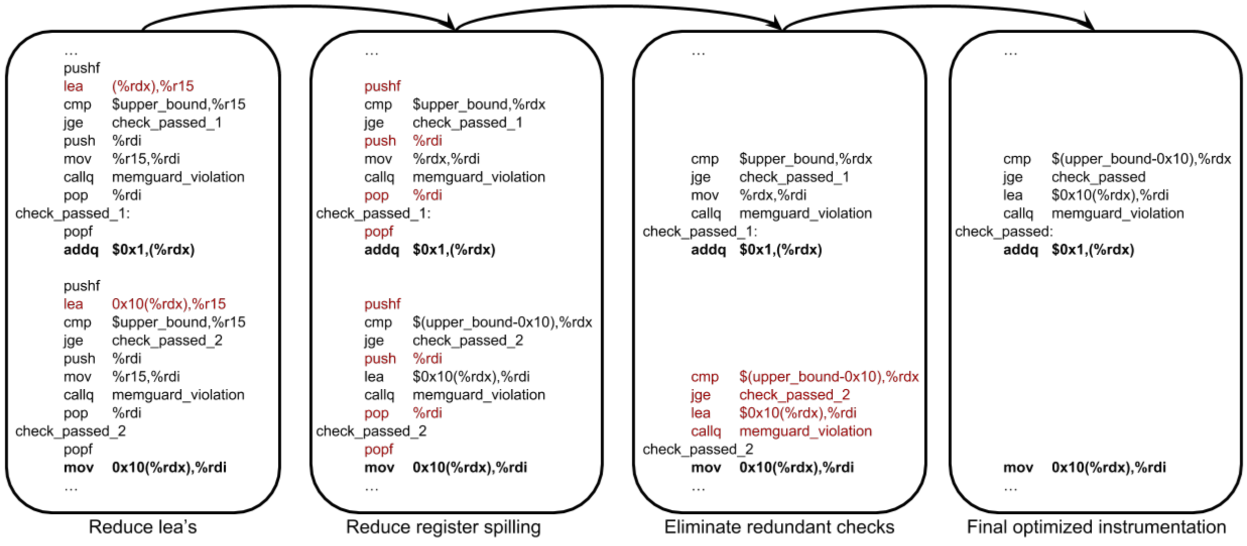
1. With MPX instrumentation enabled, the violation handler will be the bound range exceeded fault emitted by the processor to the operating system.
2. If MPX instrumentation is not enabled, the defaut violation handler is to simply call `exit(SIGSEGV)`.
3. MemGuard supports raising a user-specified signal (by default, `SIGUSR2`) on an access violation.
4. MemGuard supports calling a user-defined violation handler whose single argument is the `void *` violating address. The state of all registers will be preserved after making the function call if the program is to continue execution.

Thread execution can continue after a violation occurs if the user wishes, allowing for any number of responses such as re-randomization, data mangling/wiping, error reporting, intrusion detection, honey-potting, etc.

3.4 Optimizations

3.4.1 `%rip`- and `%rsp`-relative memory accesses

Instructions that only use the `%rip` register along with a displacement to access memory—as is very often the



```
        ...
        pushf
        lea     (%rdx),%r15
        cmp     $upper_bound,%r15
        jge     check_passed_1
        push    %rdi
        mov     %r15,%rdi
        callq   memguard_violation
        pop     %rdi
check_passed_1:
        popf
        addq    $0x1,(%rdx)

        pushf
        lea     0x10(%rdx),%r15
        cmp     $upper_bound,%r15
        jge     check_passed_2
        push    %rdi
        mov     %r15,%rdi
        callq   memguard_violation
        pop     %rdi
check_passed_2
        popf
        mov     0x10(%rdx),%rdi
        ...
```

```
        ...
        pushf
        cmp     $upper_bound,%rdx
        jge     check_passed_1
        push    %rdi
        mov     %rdx,%rdi
        callq   memguard_violation
        pop     %rdi
check_passed_1:
        popf
        addq    $0x1,(%rdx)

        pushf
        cmp     $(upper_bound-0x10),%rdx
        jge     check_passed_2
        push    %rdi
        lea     $0x10(%rdx),%rdi
        callq   memguard_violation
        pop     %rdi
check_passed_2
        popf
        mov     0x10(%rdx),%rdi
        ...
```

```
        ...
        cmp     $upper_bound,%rdx
        jge     check_passed_1
        mov     %rdx,%rdi
        callq   memguard_violation
check_passed_1:
        addq    $0x1,(%rdx)

        cmp     $(upper_bound-0x10),%rdx
        jge     check_passed_2
        lea     $0x10(%rdx),%rdi
        callq   memguard_violation
check_passed_2
        mov     0x10(%rdx),%rdi
        ...
```

```
        ...
        cmp     $(upper_bound-0x10),%rdx
        jge     check_passed
        lea     $0x10(%rdx),%rdi
        callq   memguard_violation
check_passed:
        addq    $0x1,(%rdx)

        mov     0x10(%rdx),%rdi
        ...
```

Figure 1: Optimization examples for MemGuard instrumentation and associated overheads

case when accessing global variables or data within the text section such as jump tables—cannot be controlled by an attacker. The same is true for `%rsp`-relative memory accesses, which are used to access variables on the stack.

Note that even if the attacker can control `%rsp`, she cannot reliably set it within the safe region under the assumption that she has not yet bypassed MemGuard: both the secrecy and integrity of the data within the safe region are guaranteed. As a result, she may only set it as close to the safe region as possible, namely exactly the value of `upper_bound`. Given that the attacker only has uninstrumented access to instructions that are strictly `%rsp`-relative, the attacker can in such a scenario access memory up to 2 GiB below `upper_bound`. As a result, the safe region must have a 2 GiB memory guard on its upper bound, a trivial amount of memory to reserve in the 64-bit address space.

3.4.2 Reducing `lea` instructions

Recall that memory accesses instrumented by MemGuard can have up to four distinct components: `$Disp(%BaseReg, %IdxReg, Scale)`. If only one distinct register is used, the instrumentation can be optimized by entirely eliminating the `lea` instruction and updating the `upper_bound` value in the `cmp` instruction using the single register of the memory access. As a contrived example, consider the memory access `$-0x10(%rax, %rax, 4)`. An access violation will occur if and only if `5 * %rax - 16 < upper_bound` which is solvable with respect to the minimum value `%rax` can take without triggering a fault. Assuming no violation occurs, the number of instructions executed by the optimized instrumentation reduced 87% of the time—the average percentage of single-register memory accesses.

3.4.3 Reducing `%rflags` and `%rdi` spilling

We do not need to spill `%rflags` for an instruction's instrumentation if the flags register is not live at the point of that instruction. Consider the simple case of `cmp (%rax),$0`. Because the `%rflags` register is set immediately after the instrumentation, there is no need to spill its contents onto the stack. In general, `%rflags` spilling can be eliminated around 94% of the time citekRX. The same applies for `%rdi`. Thus, we can eliminate the four instructions involved in spilling `%rflags` and `%rdi` in most cases.

3.4.4 Eliminating redundant checks

There are often several memory accesses that differ only in their displacements, such as initializing members of a struct in the heap. A memory access does not need to be instrumented if the following conditions hold true:

1. a memory access that only differs in displacement is guaranteed to be instrumented prior to the instruction in question,
2. no code sequence between the two accesses modifies the value stored in either the base or the index register of the memory reference, and
3. every control flow from the first memory access will eventually execute the second memory access.

The first of these conditions can be met if either the first instruction's basic block properly dominates the second's

Application	# Checks	Code Size Increase	% Secrecy	% Integrity
Coreutils 8.29	55,354	7.14%	73.85%	33.09%
LibreSSL 2.7.2	24,582	6.16%	78.31%	27.06%
nginx 1.14.0	22,470	12.70%	81.26%	28.23%
musl 1.1.19	7,329	5.19%	70.54%	35.22%
Average	-	7.80%	75.99%	30.90%

Figure 2: Analysis of instrumented memory accesses.

or the first instruction is executed before the second instruction within the same basic block.

The second property can be verified at compile-time by iterating through the instructions between the two memory references in question. Of importance, we must assume that any indirect branch modifies the contents of all registers: our threat model allows the attacker to control arbitrary memory outside of the safe region and thus can control the target of an indirect branch instruction.

The third property, analogous to post-domination, is important, however, in preventing false negatives in pathological cases. If we ignore this property, then it is possible to construct such an scenario: consider an offset of `-0x100000` from a base register `r` that is only used if a condition `x` is false. Propagating the instrumentation to before the check of `x` and the potential branch is semantically incorrect. It would be possible that the initial memory reference, now instrumented with a displacement of `-0x100000`, will trigger a violation despite `x` being true and the memory reference never actually being accessed. This is, in general, avoided if we know that the basic block of the second instruction post-dominates that of the first instruction.

If these three conditions hold, then the second instrumentation can be eliminated entirely. In the future, additional custom optimization may be applied to commonly-called functions with simple loops such as `strcmp` and `memset` by checking the initial address and final address that will be accessed prior to entering the loop body.

3.4.5 Intel's Memory Protection Extensions

With Intel's Memory Protection Extension (MPX) hardware feature, the instrumentation can be performed efficiently with the `bndcl` instruction [2]. This particular instruction is designed to function as a `nop` when the supplied memory address is greater than or equal to the bound in the specified bound register as it changes no flags. Otherwise, it issues a bound range exceeded fault to the operating system.

As in previous papers, we essentially misuse this feature to enforce our own security check [14]. In particular, we compare any unsafe memory access against `upper_bound` without needing to reserve a scratch register, load the effective address of the memory access, execute the `cmp` instruction, conditionally jump, or spill `%rflags` in any situation. Of course the security of this now lies on the assumption that the attacker cannot control the bounds register being used (by default, `%bnd0`). We expect this optimization to bring the average overhead of MemGuard to below 2%.

4 Implementation

To enable MemGuard, one only needs to pass the `-x86-memguard` compiler flag to the LLVM backend (see Appendix A for a listing of all compiler flags). Our implementation totals roughly 1,000 lines of C++ code targeting Linux x86-64 in the form of two LLVM passes. The first pass is at the intermediate representation level and primarily consists of adding attributes to functions that should not be inlined such as the violation handler and any function marked safe (those functions prefixed with `__safe_`. The second pass, at the machine code level, is where all the actual instrumentation occurs. In particular, we insert our instrumentation after all other optimizations have completed, immediately before the actual machine code is emitted.

Performing instrumentation immediately prior to code emission ensures two things. First, no additional unsafe memory accesses will be generated by the compiler. Second, no instruction reordering optimizations will introduce a time-of-check-to-time-of-use vulnerability if, for example, the compiler inserts an indirect branch (which our attacker can control in our thread model) between the instrumentation and memory access.

The same approach can be used to target other architectures; however, care must be taken as to which optimizations are applied. For example, all `%rsp`-relative memory accesses would need to be instrumented in an x86 architecture as the 32-bit displacement can span the entire address space. As a result, we expect software-only instrumentation performance to worsen on x86.

5 Performance

Using our MemGuard-enabled LLVM compiler, we built from source four major libraries and applications: Coreutils 8.29, LibreSSL 2.7.2, musl 1.1.19, and nginx

Application	Command	Performance
Coreutils 8.29 ls	ls -lah 10K files	1.68%
Coreutils 8.29 sort	1 GiB numbers	12.45%
LibreSSL 2.7.2	speed	1.47%
nginx 1.14.0	ab -t 60	5.23%
Average	-	5.20%

Figure 3: Performance results for MemGuard

1.14.0. In each case, we used software-only instrumentation with a safe region size of 64 MiB, not including the 2 GiB of page guards. The average overhead from the individual tests reported in Figure 3 is 5.2% when instrumenting all memory reads and writes to guarantee both the secrecy and integrity of the safe region. Additionally, the code size increase of executable and shared libraries is around 7.80%.

Generally, guaranteeing the secrecy of the safe region is more expensive than guaranteeing the integrity due to the higher frequency of memory reads compared to memory writes. If only secrecy is desired, an average of 76% of the original instrumentation is required whereas the integrity requires only 31% of the instrumentation. The percentages do not add to 100% due to the redundant check elimination optimization in some cases covering both memory reads and writes. The code size and performance metrics decrease roughly proportionally to the instrumentation.

6 Security

Using MemGuard alone, we can guarantee the secrecy and integrity of any data placed within the safe region. This does, of course, guarantee the security of any sensitive data such as cryptographic keys, password hashes, or any other information placed in the safe region, even in the presence of an attacker with access to an arbitrary read and write vulnerability. Any memory access that can possibly be attacker-controlled is verified at runtime to ensure the safe region cannot be access by the attacker.

What is more interesting from a security perspective is the use of MemGuard as a foundational tool for other defense mechanisms. Any defense mechanism that has been bypassed due to its reliance on information hiding will be hardened. In particular, the various attacks against specific code pointer integrity (CPI) implementations will fail [13, 8]. These attacks do not exploit some fundamental flaw of the underlying concepts of CPI but rather the specific implementation-dependent reliance on information hiding; the entire presumption of these attacks is that as soon as they find the location of the metadata, they can modify it so as to allow other exploitation vectors such as ROP to succeed.

If, however, a CPI implementation such as the original CPI proposal or ASLR-Guard were to place all pointer metadata within the safe region along with the integrity-only portion of MemGuard, the security guarantees initially claimed by the authors are restored [11, 12]. Specifically, no attacker will be able to hijack the control-flow of the program, assuming the attacker is not using a side-channel attack capable of accurately modifying any arbitrary memory.

Additional security mechanisms that would benefit from being coupled with MemGuard include any software-based shadow stack implementations [3], stack canaries implementations [5], as well as any safe stack implementations [1].

7 Related Work

Several works related to MemGuard have been implemented in the past, although to the best of our knowledge, none for x86-64 userland processes, and none to implement a safe region of readable and writable data.

The SFI extension to sandboxed binaries in Google's Native Client [15] with the intention of running untrusted code in a sandbox and thus required restricting stores to memory to a particular portion of the address space. The NaCl SFI extension introduced about 5% overhead for the ARM architecture versus 15% overhead x86-64. Limitations of the NaCl SFI scheme include the lack of accessing memory with two arbitrary registers: it is completely disallowed within ARM, and the base register must be one of `%rsp`, `%rbp`, or `%r15` within x86-64.

LR^2 [4], a particular execute-only memory (XoM) implementation designed for ARM processors, similarly uses bit-masking to restrict all memory reads to the data portion of the binary. The software load-masking introduced an average overhead of 6.6%. Similarly to NaCl, only single-register addressing is allowed. Although closer to MemGuard in its design goal, LR^2 is designed for 32-bit ARM machines and would incur a much higher performance penalty for x86. Furthermore, the current implementation of LR^2 has a fundamental design issue regarding relative addressing within x86-64 user memory, which spans 2^{48}. Splitting the code and data across such a large address space would require replacing all `%rip`-relative memory addressing—ubiquitous in position independent code with maximum relative distance of $\pm 2^{31}$.

Closest to MemGuard in implementation is kRˆX [14], which enforces, among other things, XoM for the kernel in an approach similar to LR^2. The kernel sections are rearranged such that all executable code and no data lies within the topmost portion of the address space, and all memory reads are compared against the lower bound of executable code. Notably, kRˆX does not require a specific address layout for the kernel, works for the x86-64 architecture, and only introduces an average of 10.86% on latency and 6.43% on bandwidth for the software-SFI instrumentation on memory reads.

8 Conclusion

We have designed and implemented MemGuard, a flexible SFI scheme that guarantees both the secrecy and integrity of a safe region in memory by instrumenting all unsafe memory loads and stores, respectively, with a low performance overhead of 5.2% and code size increase of 7.8%. Additionally, MemGuard can be used to alleviate the problems that several security mechanisms have recently faced by relying on information hiding by providing a deterministic rather than a (weak) probabilistic security guarantee.

The source code for MemGuard integrated within LLVM as described can be found at `https://github.com/Armanious/MemGuard`.

References

[1] SafeStack. `http://clang.llvm.org/docs/SafeStack.html`. Accessed: 2018-05-14.

[2] Introduction to Intel Memory Protection Extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`, July 2013. Accessed: 2018-05-14.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 340–353.

[4] BRADEN, K., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., LIEBCHEN, C., AND SADEGHI, A.-R. Leakage-resilient layout randomization for mobile devices. NDSS.

[5] COWAN, C., PU, C., DAVE MAIER, J. W., PEAT BAKKE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium* (San Antonio, TX, 1998), USENIX Association.

[6] ETOH, H., AND YODA, K. Protecting from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp`, 2000.

[7] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 475–490.

[8] GÖKTAŞ, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 105–119.

[9] GRAS, B., RAZAVI, K., ERIK BOSMAN, H. B., AND GIUFFRIDA, C. ASLR on the line: Practical cache attacks on the MMU. NDSS.

[10] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)* (Dec 2006), pp. 339–348.

[11] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 147–163.

[12] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 280–291.

[13] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 121–138.

[14] POMONIS, M., PETSIOS, T., KEROMYTIS, A. D., POLYCHRONAKIS, M., AND KEMERLIS, V. P. kR^X: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 420–436.

[15] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 1–1.

[16] SPENGLER, B. On exploiting null ptr derefs, disabling selinux, and silently fixed linux vulns. `http://seclists.org/dailydave/2007/q1/224`, March 2001.

[17] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 203–216.

[18] WILLIAMS-KING, D., GOBIESKI, G., WILLIAMS-KING, K., BLAKE, J. P., YUAN, X., COLP, P., ZHENG, M., KEMERLIS, V. P., YANG, J., AND AIELLO, W. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 367–382.

Appendix A – Supported Compiler Flags

All flags require the *-x86-memguard* prefix. The flag *-x86-memguard* itself enables or disables MemGuard instrumentation.

-mode	secrecy, integrity, both
-mmap-min-addr	target's `vm.mmap_min_addr`
-safe-region-size	size of safe region
-initialize-safe-region	should initialize safe region
-initialization-function	function to insert `mmap`/ MPX initialization
-violation-reporting	function-call, raise-signal, mpx
-violation-handler	name of violation handler
-violation-signal	value of signal to raise
-mpx-register	bnd0, bnd1, bnd2r bnd3