Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification

Jason Gionta*, William Enck†
North Carolina State University
jjgionta@ncsu.edu*, enck@cs.ncsu.edu†

Per Larsen
University of California Irvine
perl@uci.edu

***Abstract*—Software diversity has been applied to operating system kernels to protect against code-reuse attacks. However, the security of fine-grained software diversification relies on ensuring that the code layout remains secret. Unfortunately, memory disclosure vulnerabilities assist adversaries in bypassing software diversity protections by leaking the code layout. In this paper, we propose KHide, a system that thwarts kernel code-reuse attacks by combining fine-grained software diversity techniques and memory disclosure protection. First, we apply multiple fine-grained software diversity techniques to kernel code at compile time. Next, we propose a technique to protect diversified kernel code against memory disclosure at runtime. As a result, an attacker cannot predict or identify gadgets in memory to launch code-reuse attacks. We implement KHide for the Linux kernel. Our evaluation shows that KHide disclosure protection has negligible performance impact in comparison to fine-grained software diversity. We provide a security analysis of KHide calculating the survivability of gadgets across diversified versions. Our results show that KHide provides comprehensive protection against the threat of kernel code-reuse with acceptable performance impact.**

I. Introduction

Operating system vulnerabilities can enable unauthorized access to protected system resources and undermine the entire security of a system. Kernel exploits often rely on memory corruption vulnerabilities that allow an attacker to divert program flow from expected execution to chosen instructions. Modern systems have addressed the threat of code-injection and integrity attacks; however *code-reuse* attacks remain a viable option for kernel exploitation.

Code-reuse paradigms continue to evolve as new mitigations are proposed [32], [4], [13], [15]. A small number of kernel based mitigations have been deployed and detection heuristics demonstrated [29]; however these protections have known limitations that allow bypasses [21], [31], [6].

One approach to mitigating code-reuse attacks is *software diversity* [24]. Code-reuse attacks require specific knowledge of the code layout in system memory for an attacker to generate exploits. Without diversity, an attacker has a priori knowledge of the code layout of victim systems. Code randomization introduces divergence between systems, which makes reliable exploitation hard or impossible.

While software diversity is a powerful approach to prevent code-reuse attacks, code randomization is only secure to the extent that the code layout can be kept secret from attackers. Diversified systems that do not take steps to prevent memory disclosure remain vulnerable. Recent research has demonstrated that memory disclosure can be used to assist in bypassing diversity protections [31], [33], [10], [3], [30]. Snow et al. [33] showed that a single memory disclosure vulnerability can be used to systematically read diversified code and automatically generate ROP based exploits. Davi et al. [10] discussed a technique to bypass code diversity using C++ virtual method table code pointers to read code. These findings highlight the fact that memory disclosure is a pervasive threat that all diversifying defenses must address.

Our goal is to protect operating system kernels against memory disclosure vulnerabilities. In turn, we can ensure kernel code layout is not disclosed and hence protect against code-reuse attacks. Recent research focuses on protecting userspace code against the threat of memory disclosure [11], [1], [2], [10], [28]. Unfortunately, this research considers the kernel as part of the Trusted Computing Base (TCB) and hence does not protect against kernel memory disclosure.

In this paper, we propose *KHide* as a system that leverages both software diversity techniques in tandem with memory disclosure protection to harden an operating system kernel against the threat of code-reuse attacks. KHide "hides" kernel code that has been randomized. Conceptually, if code instructions reside at random locations, and memory disclosure vulnerabilities cannot read code page memory to locate code, then launching a code-reuse attack requires brute force guessing instruction locations and semantics. To accomplish this, we first apply instruction level diversification at compile time across kernel C sources and assembly files to ensure each host has a different instruction layout. Next, the KHide runtime leverages existing commodity hardware features to enable protections to disallow direct kernel code reads. We implement KHide in a widely used monolithic kernel to evaluate the feasibility of adoption and performance impact of the deployed protections. We find that KHide has limited impact on system performance (0%-10% for macro-benchmarks) and no gadgets remain at the same locations across diversified kernel versions.

This paper makes the following contributions:

- *We fix a weakness with existing kernel defenses based on diversity.* This weakness undermines previous research on operating system kernel diversification [22], [12], [18]. We propose a low impact diversification technique to address the weakness.
- *We propose memory disclosure protection for kernel code and integrate diversity protections.* Our proposed approach can be realized on commodity hardware and can also be applied to existing research on memory disclosure protections to overcome limitations for protecting

This work is supported by U.S. National Science Foundation (NSF) under grant CNS-1330553.

userspace applications.

- *We implement and evaluate KHide protections in the Linux kernel*. We find that KHide has reasonable impact on system performance. The performance impact of KHide is similar to ASLP [22] and much less than KCoFI [8]. We find that no indirect branches exist at the same location across all KHide diversified kernel images.

Finally, during our evaluation of KHide, we found that previous research applying software diversity to operating system kernels has not properly characterized their security and performance impact [22], [12], [18]. This error has led to incorrect assumptions regarding the value of software diversity for OS kernels. In contrast to prior work, we provide fine-grained diversity techniques at the instruction level along with runtime memory disclosure protection of kernel code. We discuss the limitations of prior work in Section VII.

II. Why Software Diversity?

Researchers have proposed several viable approaches for mitigating code-reuse attacks. However, applying protections to kernels is often challenging due to complex software conventions and performance requirements. For example, Code Pointer Integrity (CPI) [23] uses static analysis and type information to identify and manage code pointers; however, kernels often use opaque pointers and assembly which significantly complicates the application of CPI. Control Flow Integrity (CFI) has been applied to the FreeBSD kernel [8]. Unfortunately, the authors trade security for performance using relaxed CFI enforcement, which is circumventable [13]. In contrast, software diversity uses randomizing transformations to make the attack surface unpredictable. Furthermore, software diversity has been shown to have low performance impact [18] and provides a moving target defense strategy [24].

Software diversity has been applied to kernel code. Kernel Address Space Layout Randomization is a coarse-grained diversification technique that relocates kernel images on boot. Research has investigated medium-grained diversification of kernel code at both compile-time and run-time [22], [12]. Researchers have also demonstrated fine-grained diversification of an entire system stack using NOP insertions [18]; however, the applied diversity is not comprehensive (i.e. assembly code is not diversified).

Unfortunately, previous software diversity research fails to address the threat of memory disclosure for kernel code. Memory disclosure is a subset of information leakage that allows unauthorized access to read system memory. Leaked memory assists in bypassing software diversity protections [31], [10], [33] as well as relaxed CFI enforcement [13]. While recent research has proposed memory disclosure protections [1], [11], [7], it does not protect kernel code. KHide applies comprehensive kernel code-reuse protection by diversifying all kernel sources and preventing memory disclosure at runtime.

III. Assumptions and Threat Model

We assume the system is configured with virtual addressing via a Memory Management Unit (MMU). The system provides the ability to differentiate readable memory from executeable memory. This can be realized through Hardware Assisted Paging on x86 or on ARM.

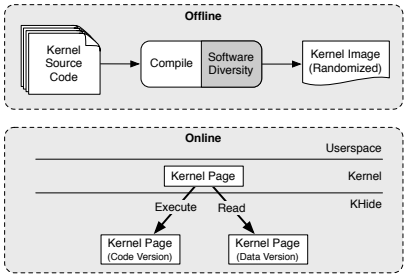

Fig. 1: Overview of KHide.

The goal of the attacker is to escalate privilege from user to kernel via code-reuse based attacks. The attacker has access to memory disclosure vulnerabilities that can read arbitrary kernel memory. However, the attacker cannot read kernel binaries from disk (i.e., only readable by root). Furthermore, the attacker cannot discover, identify, or recreate the seed used to generate the kernel image. Each kernel is uniquely compiled per deployment. Finally, an attacker cannot crash kernel execution without being identified. Therefore, we are not concerned with attacks that require a system crash to induce disclosure [3], [30].

IV. KHide

The goal of KHide is to prevent all kernel code-reuse attacks. KHide applies two broad approaches to enable protection, as depicted in Figure 1. First, KHide applies fine-grained software diversity to kernel code at compile time to create randomized kernel binaries. This technique prevents adversaries from using a priori knowledge of gadget locations to launch code-reuse attacks. Therefore, the adversary must resort to accessing kernel memory to discover gadgets. Next, KHide prevents gadget discovery by ensuring memory disclosures cannot read kernel code. We prevent reading of kernel code by enforcing a *code reading* policy [11] through hardware based permissions. Reads of kernel code are trapped by KHide and redirected to readable kernel data. As a result, an attacker cannot identify or find gadgets required for bypassing fine-grained instruction diversification.

KHide generates unique kernel images per compilation using fine-grained instruction diversity. Specifically, we adopt a technique proposed by Homescu et al. [14], which inserts NOPs at a configurable frequency between machine instructions. While NOP insertion randomizes potential gadget layouts in memory, we observe a weakness: code layout can be inferred from disclosed return addresses on the stack. We enhance Homescu et al.'s approach by (1) moving call instructions to the beginning of functions, and (2) randomizing register allocation. Moving call instruction locations decouples return addresses from the function layout. Randomizing register allocation prevents an attacker from using a priori knowledge of a gadget following a call instruction.

Fine-grained diversification often becomes ineffective in the presence of memory disclosure [33], [10], [30]. Previous proposals [1], [11] protect against memory disclosure; however, they configure and mediate memory access using the operating system kernel. Therefore, they are not suitable for protecting kernel code itself.

KHide mediates read access to kernel code using existing hardware features. Physical memory pages containing code are marked as execute-only and therefore not readable. Our implementation of KHide leverages Hardware Assisted Paging (HAP) to enable execute-only memory on modern x86 platforms. HAP is a general term to represent recent virtualization extensions that allow the hardware to map guest physical to host physical addresses (e.g., EPT on Intel). Execute-only permissions are possible for guest-to-host mappings. KHide disables read access to kernel code during system boot after memory initialization. At this time, the kernel registers physical pages for protection. The registration marks the code pages as execute-only and hence the code is no longer readable.

Unfortunately, there are legitimate reasons for code pages to be read during execution. Our experiments identified two such cases: (1) kernel function tracing (i.e., kprobes and ftrace) and (2) embedded data (i.e., jump tables). To address these requirements, we define a code reading policy [11] to allow selective reads to shadow pages. Shadow pages only contain the necessary data to allow for correct execution. KHide then disallows execution of data in the readable shadow page.

Executable kernel code is not limited to the kernel image. For example, loadable kernel modules contain executable code. To protect kernel modules, we modify the loading process to (1) separate kernel code from data, and (2) register kernel module code pages after module initialization.

V. Design and Implementation

The design of KHide is generically applicable to any modern operating system kernel that uses virtual memory to enforce permissions. To simplify discussion, we describe KHide in context of the Linux kernel and x86 processors supporting Hardware Assisted Paging.

A. Kernel Code Diversification

There are two general approaches for applying software diversity: (1) binary rewriting and (2) compiler-based transformations. Binary rewriting requires runtime knowledge of machine code execution and is primarily used for legacy software. Compiler-based randomizations allow complex transformations by leveraging high level information in source code available during compilation. However, compiler-based transformation requires source code access, which prevents diversification of binary-only, legacy applications.

Rewriting binary kernels is challenging due to their use of interleaved performance critical machine code (e.g., 16-bit, 32-bit) and significant use of function pointers [24]. Kil et al. [22] used binary-rewriting to shuffle function locations of the Linux kernel. However, given the availability of kernel source code to developers and manufacturers, compiler-based diversification is better suited for kernel code diversity.

1) Compile-Time Transformation: There are two popular compilers that provide extensible architectures for building diversification: GNU Compiler Collection (GCC) and LLVM. Previous software diversity research has used both compilers [19], [14], [12]. However, LLVM is generally regarded as more flexible for building analysis and optimizations.

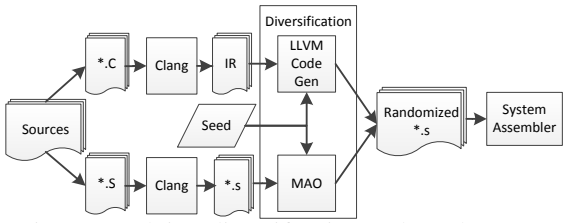

Fig. 2: Applying diversification to kernel sources.

Kernel code diversification has two challenges not typically present in userspace applications:

C-1: Kernel code is often highly optimized and therefore sensitive to modification.

C-2: Kernel code contains target dependent assembly files that may make assumptions about code layout and structure.

To address **C-1**, we adopt an approach proposed by Homescu et al. [14] that inserts NOP instruction at a given probability between machine instructions. Insertions are based on a seed, which allows for regeneration of kernel images. As a result, potential gadgets change in both size and location. Homescu et al. provide a modified LLVM/Clang tool for NOP insertions. This tool is invoked at code generation time as an LLVM *machine function pass*. This placement allows control over machine/target specific instructions and does not affect the generation of the LLVM Intermediate Representation (IR). Furthermore, the approach has been shown to have limited impact on performance [14].

Unfortunately, LLVM treats assembly files as machine code (**C-2**). Therefore, they are never processed by LLVM's pass infrastructure. Assembly files are either passed directly to the system assembler (e.g., portable GNU assembler) or through the LLVM integrated assembler. The LLVM integrated assembler directly produces machine code without performing the analysis required to perform optimizations.

We address **C-2** by applying an assembler-specific diversification as depicted in Figure 2. We leverage the Micro-Architectural Optimizer (MAO) [16] that enables optimization of x86 assembly via source-to-source transformations.

MAO NOP Insertion Pass: We implement the NOP insertion as a MAO optimization plugin. We created a *MaoFunctionPass* that enumerates each identified function and assembly instruction. NOPs are inserted prior to assembly instructions based on a provided seed and configurable insertion probability/rate.

Figure 2 depicts the overall kernel compilation process. C sources are compiled with LLVM/Clang and diversification is applied during code generation. The output of code generation is a diversified assembly file that is fed to the system assembler. Preprocessed assembly source files (i.e., `*.S`) are passed through Clang and provided to MAO as assembly files (i.e., `*.s`). MAO applies our diversification, producing assembly files with randomly inserted NOPs.

Limitations of NOP Insertion: We note that NOP insertion alone is not sufficient when return addresses are leaked. For example, a leaked stack can be used to identify function return addresses and therefore determine gadgets that follow the return. Figure 3b shows a concrete example of this problem. At address 0xFF10, *printk* is called and the return address

```
; function start
0xF000: push rbp
        ...
0xFF10: call 0x5400
              ; printk
0xFF13: pop rax
        pop rbx
        pop rbp
        ret
```

(a) No diversity

```
; function start
0xF000: push rbp
        ...
0xFF10: call 0x5400
              ; printk
0xFF13: nop
        pop rax
        nop
        pop rbx
        pop rbp
        ret
```

(b) NOP Insertion

```
; function start
0xF000: push rbp
        ...
0xFF10: call 0x5400
              ; printk
0xFF13: nop
        pop rsi
        nop
        pop rdx
        pop rbp
        ret
```

(c) Register Randomization

```
; function start
0xF000: nop
        jmp 0xF100
0xF004: call 0x5400
              ; printk
        jmp 0xFF63
    ...
0xF100: push rbp
    ...
0xFF60: jmp 0xF004
0xFF63: nop
        pop rsi
        nop
        pop rdx
        pop rbp
        ret
```

(d) Full Diversity

Fig. 3: Instruction Diversification Examples

(0xFF13) is pushed on the stack. If the return address is leaked, the attacker can determine the location of the gadget (i.e., pop rax, pop rbx, pop rbp, ret) following the return and can guess locations of additional gadgets with high probability (i.e., pop rbx, pop rbp, ret).

2) Hardening Diversity Against Stack Disclosure: To address the limitations of diversity via NOP insertion, KHide provides two enhancements: (1) register selection randomization and (2) call site lifting. Together, these techniques modify the semantics of potential gadgets and their proximity to return addresses. In doing so, KHide prevents an attacker from misusing or guessing gadgets that follow a call instruction, thereby protecting against an adversary that finds return addresses on the stack. First, register selection randomization modifies registers used in instructions. Figure 3c enhances our example with register randomization. The *rsi* and *rdx* registers are swapped for *rax* and *rbx*, respectively. In some cases, registers cannot be randomized due specific instruction requirements. For example, *rbp* is used for stack frame management. Therefore, given a leaked return address, an attacker can guess with higher probability the location of instructions that use static register assignment. Call site lifting is a novel diversification strategy that decouples the return address from the instructions following a call. Call instructions are moved to the beginning of functions and the order is randomized. This modification preserves the order of potential performance sensitive code in the function and leverages code locality by executing the call instruction within the diversified function. Figure 3d shows code after call site lifting is applied.

Register Randomization: Register assignment occurs as the LLVM IR is transformed (i.e., lowered) into *machine instructions*. At this time, LLVM creates virtual registers that are associated with operand usage; the virtual registers are later assigned to physical registers. Applying register randomization at this mapping step seems intuitive; however, instructions may require specific register operands. For example, the x86 32-bit divide instruction requires the use of eax and edx as operands. To address these dependencies, the LLVM backend abstracts specific registers into register classes based on operational use.

We achieve register randomization in LLVM by modifying the preferred order of register class sets. LLVM register allocators request the preferred assignment order for specific classes using *RegisterClassInfo*. We randomize the preferred register order for each class based on the provided seed. Register classes have a *cost per use* based on callee saved registers. We shuffle the preferred order with respect to costs, which is then used for virtual register allocation.

Call Site Lifting: Call site lifting is similar to basic block randomization. However, algorithms that create basic blocks treat call instructions as terminating. In contrast, call site lifting considers both the instruction before the call instruction and the call instruction itself as terminating. As a result, each call site is isolated into its own basic block. By creating a new basic block containing only the call instruction, the basic block can then be moved and shuffled independent of the execution flow.

Basic block randomization through LLVM can potentially be implemented using the LLVM IR and applied as an LLVM Link Time Optimization (LTO). The LLVM IR contains *call* instructions that represent target independent code. However, applying call site lifting on the LLVM IR is problematic. First, call site lifting can be undone at machine code generation by optimizations that attempt to remove unnecessary branches. Second, the LLVM IR instructions map to multiple target specific instructions that include an epilogue which is known to directly follow a call instruction in memory.

Instead, we implement call site lifting as an LLVM *MachineFunctionPass* module that runs over each function to identify, isolate, and shuffle target specific (e.g., x86) call instructions. The pass iterates over all machine instructions to identify calls. For each call, we create one *machine basic block* to hold the call instruction and a second *machine basic block* to hold all of the instructions following the call. We must take care to update the original basic block and call instruction block with branching instructions to preserve correctness. After all calls are moved to isolated *machine code blocks* with correct branching and successors, we move the calls to the beginning of the function and shuffle based on the seed. Moving the call blocks within a function still benefits from code locality. Finally, we create a new function entry block to force calls to remain directly after the function entry in memory.

3) Diversifying the Linux Kernel: Figure 2 provides a general compilation process for diversifying C/C++ sources as well as assembly sources of a kernel. This flow can be applied to other kernel build processes. However, we focus on application to the Linux kernel. Traditionally, the Linux kernel is compiled with the GCC compiler. However, a recent project called LLVMLinux allows the Linux kernel to be built with LLVM/Clang. This ability provides new opportunities for enabling analysis and optimizations not easily achieved with GCC. LLVMLinux requires Clang version 3.5 or higher, which enables support for compiling 16-bit code on x86 based platforms. We forward ported the software diversity tool published

by Homescu et al. [14] to LLVM/Clang 3.6 to enable support of the latest versions of LLVM/Clang and LLVMLinux 3.18. We fixed several bugs and informed the LLVMLinux team of the issues. Finally, LLVM code diversification is enabled and seeded via compiler flags. Assembly files are passed through a wrapper script that invokes the MAO diversification pass.

B. Kernel Memory Disclosure Protection

Memory disclosure vulnerabilities can allow adversaries to *read* randomized code, thereby allowing them to bypass fine-grained diversity protections [33], [30], [10]. For example, Seibert et al. [30] outlined a side channel leak of memory using a data pointer overwrite to read code.

To protect against kernel memory disclosure, KHide prevents kernel code from being read. By making kernel code *execute-only*, memory disclosures cannot leak information about diversified code by directly reading instructions. Recent hardware additions enable execute-only memory on the x86 and ARM. On x86 platforms, execute-only memory permissions can be enforced through Hardware Assisted Paging (HAP). We discuss HAP in Section V-B1.

In some cases, data in executable pages must be read during normal execution. For example, jump-tables are embedded into position independent code to support switch statements. Similarly, code may be read to enable optimizations or tracing.

To address this challenge, KHide uses a *code reading policy* to apply fine-grained read permissions on executable memory. This approach was previously proposed by Gionta et al. [11] to allow for selective reads of code for userspace applications. At a high-level, the data that requires reading from executable pages is copied to readable *shadow pages*. CPU read access is directed to the shadow read-page. CPU instruction fetches are made to the diversified code-page.

Unfortunately, Gionta et al.'s code reading policy technique does not apply directly to kernel code. First, they required a split-TLB architecture to differentiate execute and read operations. Modern hardware uses a unified L2 TLB architecture; therefore a split-TLB cannot be used to differentiate CPU operations. Second, Gionta et al. uses the kernel to enforce code reading policy. Therefore, kernel code is part of the Trusted Computing Base and assumed free of memory disclosure vulnerabilities.

KHide overcomes these challenges by enforcing policy outside of the kernel and userspace (e.g., in a hypervisor). Therefore, KHide code reading protections cannot be modified or bypassed. Code reading policy is created by dividing data that may be read on code pages into separate shadow readable pages. The policy is enforced by mediating accesses on protected pages. That is, code pages are marked execute-only and shadow read pages are read-only. KHide manages permissions on protected pages by trapping access violations and mapping the appropriate page based on operation type (i.e., execute, read). Figure 4 depicts this enforcement.

1) Execute-only Memory with HAP: Hardware Assisted Paging (HAP) is a general term that represents a feature of hardware virtualization extensions that map guest physical addresses to host physical addresses. Different hardware

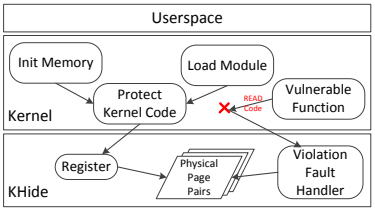

Fig. 4: Disclosure protection by enforcing code policy.

manufacturers have platform specific names (e.g., EPT on Intel, NPT on AMD). HAP alleviates a significant amount of host interaction required by classical virtualization memory management (i.e., shadow paging) [17].

Similar to traditional memory management, HAP uses page-tables entries to set permissions and manage the status of physical memory. However, HAP supports explicit read permissions. HAP represents a non-present page by clearing bit-0 (R), bit-1 (W), and bit-2 (E) of page-table entries [17]. Execute-only memory is enabled on physical pages by setting a HAP entry bit-2 (i.e., execute permissions) and clearing bit-0 and bit-1 (i.e., read and write permissions).

2) Kernel Code Reading Policy: HAP configuration enables execute-only permissions on a page granularity. Unfortunately, code pages may contain data that needs to be read for correct execution. Therefore, naïve use of execute-only permissions is too restrictive.

KHide provides selective fine-grained reads of code pages by applying code reading policy. The policy identifies pages containing both executable and read-only data. KHide enforces policy by mapping code parts as execute-only and data parts as read-only. Policy creation involves dividing readable and executable parts into separate physical memory pages.

Identifying Data to Read: At compile time, KHide identifies data on code pages that needs to be read. This data includes two types: (1) jump tables and (2) machine code (i.e., instructions). Jump-tables are read-only data embedded in executable pages to handle switch statements in position independent code (PIC). As we discussed earlier, machine code may be legitimately read by tracing tools (e.g., kprobes, ftrace).

To allow reads of jump-tables in kernel code, we instrumented LLVM/Clang to track embedded jump-tables. Specifically, we modified the LLVM ASM printer responsible for outputting assembly to track jump-tables emitted to code sections. Jump-table locations and sizes are written to the kernel image and modules as read-only data.

Kernel code must also be read to support tools that profile execution. For example, ftrace and kprobes read the first byte of a target function and write an *int3* in place. In turn, the traced functions will trap on execution and the tool proceeds to replace the first byte with the originally read value. Functions may be located at different byte offsets and pages. One approach to support tracing tools with KHide is to copy the first instruction of each function to the shadow read-only page. Supporting this design raises two concerns. First, policy creation requires knowledge of each function location to copy bytes to the shadow read page. Second, one byte for each function will be disclosed by placing it on the shadow read

page and therefore function locations can be disclosed by identifying these bytes on read-only pages.

Instead, KHide supports kernel profiling tools by making all functions share the same first byte. Specifically, a NOP instruction is added to the start of each function. Since all functions share the same instruction, this approach simplifies policy creation and prevents information leakage to disclosed bytes on read-only pages. We insert a single NOP instruction for each function as part of the *call site lifting* diversity pass. Recall that during compilation, we create a new *MachineBasicBlock* for each function and insert it as the entry block. During this process, we add a single NOP instruction to the entry block. By default, read-only pages are filled with NOP instructions at policy creation time. Any code page reads will return a NOP instruction.

Policy Creation and Initialization: At runtime, KHide (1) identifies readable data in code pages and (2) copies that to the read-only page. The code reading policy creation consists of creating pairs of code pages and associated readable shadow pages. We discuss policy enforcement in Section V-B4.

Policy creation can occur (a) implicitly, e.g., using kernel metadata and load information, or (b) explicitly, e.g., the kernel provides runtime information. Implicit policy creation allows for transparent disclosure protection without kernel modification. However, it also hinders flexibility, prevents extending protections to userspace, and allows for potential disclosure of code via a time-of-disclosure-to-time-of-protection vulnerability. Therefore, we use explicit policy creation in our implementation of KHide.

For explicit policy creation, the KHide-aware kernel registers code pages for disclosure protection after memory is allocated and permissions are set (i.e., non-writable, non-executable). At this time, the KHide-aware kernel identifies readable data on allocated code pages using embedded data locations added during compilation. For kernel images, information is located using an exported symbol at system boot as part of memory initialization. For kernel modules, data locations are read from a non-loaded section of the binary at load time after the code sections have been copied to memory and marked up for relocation. By default, the shadow read page is filled with NOP instructions to support function tracing. Data to be read on code pages is moved to readable pages allocated by the kernel. Moving of data prevents misuse of jump-tables located on code pages. The pairs of code and shadow data pages constitute the code reading policy. In some cases, code pages will not have data to read and can share a common read-only shadow page to reduce the memory footprint.

3) Policy Registration: Once the policy is created, the kernel must explicitly register the pairs of code and shadow read pages. This process informs KHide that memory has been laid out, configured, and ready for disclosure protection. Policy registration is the process of providing the physical addresses of code and shadow read page pairs via hypercalls.

Upon registration, KHide uses the kernel physical address of the code page to look up previous registrations. Code pages can be registered only once and cannot be mapped readable. KHide generates two HAP page-table entries corresponding to the code page and the read-only shadow page. The code page entry is set as execute-only and the shadow read page entry is set as read-only. These entries are saved and indexed using the code page physical address.

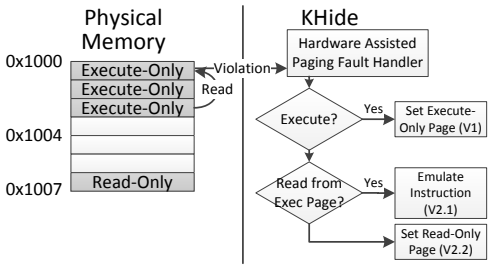

Fig. 5: KHide Policy Enforcement

Ideally, once a code page is registered with KHide, protections cannot be removed. This property prevents attempts to turn off protections. Unfortunately, kernels may require unmapping or freeing executable pages from memory. For example, module loading unmaps initialization code after execution. Therefore, we must support releasing unneeded code pages. KHide handles two types of registrations: *one-time registration* and *transient registration*. Each code page physical address is associated with a registration type.

One-Time Registration: One-time registration locks KHide protection until system reboot. It is used for kernel code and modules that are never moved or unloaded during execution.

Transient Registration: Transient registration allows specific memory to be later unregistered. It is used for protecting kernel module initialization code and enables support of userspace memory disclosure protections.

4) Enforcing Policy for Disclosure Protection: KHide marks registered code pages as execute-only and readable pages as read-only. During execution, the kernel will read or execute KHide protected memory. Reading execute-only memory or executing read-only memory will cause a HAP violation. Specific fault information is provided as an *exit qualification* that contains (1) the type of operation, (2) the HAP permissions set on the physical page, and (3) the guest kernel address that caused the violation.

KHide enforces the code reading policy by mediating access on HAP violation. Figure 5 depicts the fault handling process. Violations associated with KHide pages are categorized as either (V1) executing a read-only page or (V2) reading an execute-only code page. For execute violations (V1), KHide maps in the corresponding code page with execute-only permissions. There are two cases for read violations: (V2.1) reading from the currently executing page, and (V2.2) reading from an execute-only page that is currently not executing. At the time of violation, KHide differentiates between (V2.1) and (V2.2) by comparing the kernel instruction pointer with the faulting page (e.g., CR2 register). If the upper bytes of the values are equal, it is a read on the currently executing page. For (V2.2) violations, KHide simply maps in the corresponding read page with read-only permissions.

Special care must be taken to handle (V2.1) violations. One possible approach to handling (V2.1) is to (1) copy the

instruction from the code page to the read-only page, (2) map the read-only page as executable, and (3) single step execution over the instruction. Unfortunately, this design can allow for code injection attacks. Instead, KHide emulates the faulting operations using the execute page to parse the instruction and the read page as the data source. Therefore, KHide does not have to modify kernel memory to handle (V2.1) violations.

VI. Empirical Evaluation

We evaluate the performance impact of KHide using both micro-benchmarks and application benchmarks. We also report the memory and disk overheads for code diversity and memory disclosure protections. Finally, we provide a security evaluation that considers the changes in indirect branch locations after applying code diversity.

A. Experimental Setup

We implemented KHide code-reuse protections on Linux kernel 3.18 with LLVMLinux patches applied to support compilation with LLVM/Clang 3.6 for x86/64 platforms. We compiled the diversified kernel with a configuration that inserts NOP instructions at a probability of 50% for all instructions. This probability can be lowered to reduce performance and spatial impact [14]. KHide memory disclosure protection was built on top of KVM for the Linux kernel 3.14.17. Both Linux kernels were run on Ubuntu 14.04 LTS server edition. We implemented the software diversity techniques on LLVM/Clang 3.6 and Micro-Architecture Optimizer version 0.2.

We ran our evaluation experiments on a Dell Optiplex 9010 workstation with an Intel Core i7-3770 processor at 3.4 GHz with 8MB of cache, 16 GB of RAM, an integrated Intel 82579LM Gigabit Ethernet card, and a 7200 RPM 6 Gb/s 500 GB SATA hard drive. For network experiments, we used a dedicated isolated Gigabit Ethernet network. The client machine was a Dell Latitude E6510 with a 2-core Intel Core i5-540M at 2.53 GHz with 4 GB of RAM. The KHide protected kernel was run in a virtual machine configured with one virtual CPU and 4 GB of RAM. We used only one virtual CPU as the applied LLVMLinux patches do not have stable support of SMP. We used a TAP device to provide network access.

B. Performance Evaluation

We evaluated the performance impact of a KHide protected system by running a series of four different applications. We ran experiments under three different configurations to isolate the costs of (1) applying diversification to kernel code, and (2) enabling memory disclosure protections. We use a LLVM/Clang compiled Linux with no diversification as the baseline deployment. We use a random seed generated from `/dev/urandom` to diversify the Linux kernel. This results in different diversified versions based on the seed and may impact performance between versions. To account for this, we generated five diversified kernels for our experiments. The presented results for diversification and KHide are the average of the five diversified kernels to account for the variations in performance. For each experiment, we verified the 95% confidence interval is less than two orders of magnitude smaller than the reported mean. Therefore, we are confident the experiments are repeatable and provide a fair representation of performance impact.

TABLE I: LMBench Results: Time in Microseconds

Test	Baseline	Diversified		KHide	
null syscall	0.0352	0.0388	10%	0.0384	9%
open/close	0.777	1.191	53.3%	1.17	50.7%
page fault	0.139	0.153	10%	0.151	8.9%
sig. handler install	0.101	0.117	15.2%	0.116	14.6%
sig. handler delivery	0.622	0.749	20.4%	0.743	19.6%
fork + exit	77.29	93.83	31.4%	92.8	20%
fork + exec	81.38	100.4	23.4%	99.38	22.1%
select	4.7	6.87	46%	6.84	45.4%

We compare our results to previously published research towards kernel code-reuse protection. To highlight them: ASLP [22] performs function level randomization of Linux kernel code, Jackson et al. [18] apply NOP insertions to an entire Linux software stack excluding assembly resources, KCoFI [8] and VirtualGhost [9] enforce control flow integrity for the FreeBSD kernel.

1) Micro-Benchmark: We used LMBench [27] to investigate the impact of KHide protections on operating system operations. We installed the LMBench package from the Ubuntu repository and ran *lmbench-run* to create a standard configuration. We ran the benchmark 30 times for each of the five diversified and native (LLVM/Clang) compiled kernels.

Table I shows the average times for system operations in microseconds with overhead percentages. The overhead of KHide is similar to that of the diversified kernel. This result suggests there is negligible runtime overhead for enforcing memory disclosure protection. In fact, KHide appears to have a small non-negligible increase in performance. This improvement is an artifact of hardware that short-circuits DTLB management of memory identified as execute-only. The highest impact appears for file open operations, which has a 52% overhead. The lowest impact is 9% for null system calls and page faults. The average creation overhead was measured at 34%, with the highest being 36% for 1 KB files. The average deletion overhead was measured at 34%, with the highest being 35% for 1 KB files. Our observed impact are much more reasonable than related security research. VirtualGhost has significantly higher overhead across all LMBench dimensions, reporting a file open overhead of 483% [9]. KCoFI reported generally much higher overhead for all dimensions, including 247% overhead for file open, but a similar page fault overhead of 11% [8]. ASLP reported a fork slowdown of 12% [22], whereas KHide sees a slowdown of 22%.

2) Macro-Benchmark: We installed Apache webserver version 2.4.7 and served files containing random data sized from 1 KB to 1 MB. We ran ApacheBench version 2.3 and simulated 32 clients making 10,000 requests as an experiment. We ran each experiment 20 times for each file.

Figure 6 shows the mean bandwidth performance and 95% confidence intervals as error bars for each of the file sizes. The observed average overhead was 4% across all experiments. We measured the highest overhead of 9% for 16 KB files with minimal overhead for small files (2%-4%) and large files (0%-3%). In comparison, ASLP saw a 14% overhead; however, ASLP configuration simulated 100 clients [22]. The authors of

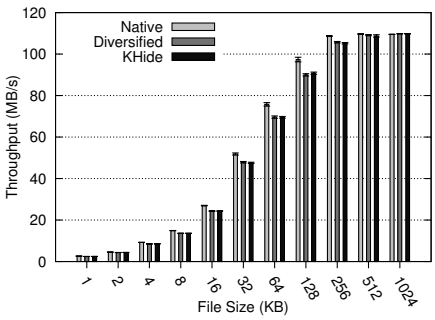

Fig. 6: ApacheBench results.

KCoFI used the same ApacheBench configuration but reported no overhead. However, KCoFI did not use Apache to serve files [8]. Jackson et al. reported similar performance impact of 10% when transferring a 15.4 KB image with NOPs inserted with a probability of 40% [18].

3) Disk and Memory Overhead: The diversified bzImages averaged 6.3 MB compared to 5.1 MB for the undiversified image (i.e., 23% increase). In memory, the diversified code averaged 10 MB compared to 7.9 MB undiversified (i.e., 27% increase). The code reading policy requires a shadow read-only page for each code page pair. Shadow read pages can be shared for code pages that do not contain data that is read. In our evaluation, we found that the Linux kernel does not embed any jump-tables in code pages and can be supported using one shared read-page (4 KB) filled with NOP instructions.

C. Security Evaluation

To assess the security of the software diversity techniques used by KHide, we evaluated the survivability of gadgets across the generated kernels. Homescu et al. [14] previously discussed gadget survival and proposed an algorithm for calculations. At a high-level, the algorithm identifies all memory addresses across diversified binaries that have identical gadget semantics accounting for NOP instructions. As Homescu et al. demonstrated, existing published ROP exploits fail when gadget locations have changed. We used the algorithm to calculate the survivability of gadgets across the five diversified kernels used in the performance evaluation.

We found that no gadgets survived across all the five diversified kernels. This result confirms that the applied diversity techniques are correctly randomizing kernel code; and as a result, an adversary must guess gadget locations. Furthermore, adversaries cannot pre-generate diversified images in attempts to find common gadgets across images. When comparing diversified kernels, we found that 6,258 gadgets survived at least two kernels, 116 gadgets survived at least three kernels, and two gadgets survived at least four kernels.

VII. Related Work

Control Flow Integrity: CFI has been proposed and modeled to enforce expected execution of code. KGuard [20] protects against return to user (ret2user) attacks that branch to userspace memory and is identical to Supervisor Mode Execution Prevention (SMEP) [17] deployed in modern x86 hardware. However, these protections do not address reuse of kernel code. Criswell et al. [8] present an approach towards enforcing CFI for the FreeBSD operating system kernel using relaxed policies. However, researchers have shown that relaxed CFI policies can be bypassed to enable code-reuse attacks in CFI protected systems [13].

Software Diversity: Research has investigated fine-grained diversification of kernel code at both compile-time and runtime. Kil et al. [22] proposed ASLP for medium-grained randomization of the Linux kernel by shuffling functions in memory using relocation information. However, gadgets within functions remain static after diversification and are therefore recognizable based on leaked function pointers or return addresses. Giuffrida et al. [12] applied software diversity techniques to a specialized microkernel with a focus on software re-randomization. They claim instruction level diversification; however, they only shift function code starting offsets. Therefore, similar to medium-grained diversification, an attacker can predict function gadget locations from leaked return addresses. Furthermore, the authors use SPEC CPU2006 for performance evaluation, which is not intended to stress networking, operating systems, or system I/O [34]. Jackson et al. [18] applied diversification of an entire system stack. However, NOP insertions were not applied to assembly files.

Memory Disclosure: Recently, memory disclosure has gained significant attention from security researchers and attackers wishing to bypass advanced system protections. Snow et al. [33] used disclosed executable data to systematically read all process memory to build code-reuse exploits and circumvent fine-grained software diversity. Davi et al. [10] used leaked pointers from C++ virtual method table to read code. Seibert et al. [30] discussed side channel leakage of code using various techniques to identify gadgets in memory. However, disclosure of fine-grained instructions required reading of code pages or crashing of applications. Memory disclosure has also assisted in bypassing relaxed Control Flow Integrity (CFI) enforcement. Göktaş et al. [13] relied on a memory disclosure vulnerability to find function call and return stubs in trampoline code to build an ROP exploit against CFI protections. Isolated execution environments (i.e., Flicker [26], TrustVisor [25], Intel SGX) also fall victim to memory disclosure.

Recent research has proposed protections focused on protecting userspace applications against memory disclosure. Backes et al. [1] propose a primitive called Execute-no-Read (XnR), which limits code page reads to a limited set of executable pages in a "sliding window." The security of XnR relies on the "sliding window" being small (i.e., contains a limited number of pages), unpredictable, and code pages not containing data to read. Unfortunately, attackers can identify the "sliding window" from return addresses on the stack and therefore leak recently used code pages without detection. Furthermore, XnR does not support selectively reading data in code pages (e.g., C++ exception handling), but rather simply allow all reads of code and apply a threshold based heuristic for detection. Gionta et al. [11] propose HideM to provide memory disclosure protection by applying a *code reading* policy based on executable data that may be read; a policy is enforced using the split-TLB to differentiate read accesses from instruction fetches on memory. However, modern hardware contains L2 unified-TLB caches, which will break the proposed technique. Mohan et al. [28] use segmentation to keep CFI flow secret

and leverages binary rewriting for randomization. However, kernel disclosure vulnerabilities can be used to leak segmentation information. Davi et al. [10] propose Isomeron, which randomizes execution flow at branches. However, Isomeron instruments all branches causing significant performance overhead. Finally, Crane et al. [7] propose Readactor to protect against memory disclosure of code pointers by redirecting calls and indirect branches through execute-only trampolines. Therefore, disclosed code pointers cannot be used to infer gadget locations. However, Readactor fails to support software that requires reading code instructions at runtime, which is supported by KHide. LR2 is a software implementation of memory disclosure protection for ARM platforms [5].

VIII. Conclusion

In this paper, we presented the design and implementation of KHide to protect operating system kernels against code-reuse attacks. We applied fine-grained software diversity techniques to prevent attackers from assuming binary layout of kernel code in memory. We proposed a novel diversity technique to address previously unidentified disclosure threats. Then, we applied runtime memory disclosure protections to defend against leakage of diversified code. We implemented KHide using the LLVM/Clang compiler and Micro-Architectural Optimizer to apply software diversity techniques to the Linux kernel. We performed an evaluation of KHide protections. We observed limited impact to system performance and no gadgets survived diversified kernel versions. Thus, KHide is a suitable solution for protecting systems against code-reuse attacks.

References

[1] Backes, M., et al. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (2014).

[2] Backes, M., and Nürnberger, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. *USENIX Security Symposium* (2014).

[3] Bittau, A., et al. Hacking blind. In *IEEE Symposium on Security and Privacy* (2014).

[4] Bletsch, T., et al. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011).

[5] Braden, K., Crane, S., Davi, L., Franz, M., Larsen, P., Liebchen, C., and Sadeghi, A.-R. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network & Distributed System Security Symposium (NDSS)* (Feb. 2016).

[6] Carlini, N., and Wagner, D. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium* (2014).

[7] Crane, S., et al. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy* (2015).

[8] Criswell, J., et al. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy* (2014).

[9] Criswell, J., et al. Virtual Ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[10] Davi, L., et al. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)* (2015).

[11] Gionta, J., et al. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the fifth ACM conference on Data and application security and privacy* (2015).

[12] Giuffrida, C., et al. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium* (2012).

[13] Göktaş, E., et al. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy* (2014).

[14] Homescu, A., et al. Profile-guided automated software diversity. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization* (2013).

[15] Hund, R., et al. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium* (2009).

[16] Hundt, R., et al. Mao–an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2011).

[17] Intel. Intel architectures manual. *Volume 3A: System Programming Guide, Part 1* (64).

[18] Jackson, T., et al. Diversifying the software stack using randomized nop insertion. In *Moving Target Defense II*. Springer, 2013.

[19] Kc, G. S., et al. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (2003).

[20] Kemerlis, V. P., et al. kguard: Lightweight kernel protection against return-to-user attacks. In *USENIX Security Symposium* (2012).

[21] Kemerlis, V. P., et al. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium* (2014).

[22] Kil, C., et al. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *the 22nd Annual Computer Security Applications Conference* (2006).

[23] Kuznetsov, V., et al. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[24] Larsen, P., et al. Sok: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014).

[25] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010).

[26] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev. 42*, 4 (Apr. 2008).

[27] McVoy, L. W., , et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference* (1996).

[28] Mohan, V., et al. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security* (2015).

[29] Pappas, V., et al. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security* (2013).

[30] Seibert, J., et al. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[31] Serna, F. J. The info leak era on software exploitation. *Black Hat USA* (2012).

[32] Shacham, H. 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* (2007).

[33] Snow, K. Z., et al. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy* (2013).

[34] Standard Performance Evaluation Corporation. *Spec CPU2006 Readme*.