HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities *

Jason Gionta
NC State University
jjgionta@ncsu.edu

William Enck
NC State University
whenck@ncsu.edu

Peng Ning
NC State University
pning@ncsu.edu

ABSTRACT

Memory disclosure vulnerabilities have become a common component for enabling reliable exploitation of systems by leaking the contents of executable data. Previous research towards protecting executable data from disclosure has failed to gain popularity due to large performance penalties and required architectural changes. Other research has focused on protecting application data but fails to consider a vulnerable application that leaks its own executable data.

In this paper we present HideM, a practical system for protecting against memory disclosures in contemporary commodity systems. HideM addresses limitations in existing advanced security protections (e.g., fine-grained ASLR, CFI) wherein an adversary discloses executable data from memory, reasons about protection weaknesses, and builds corresponding exploits. HideM uses the split-TLB architecture, commonly found in CPUs, to enable fine-grained execute and read permission on memory. HideM enforces fine-grained permission based on policy generated from binary structure thus enabling protection of Commercial-Off-The-Shelf (COTS) binaries. In our evaluation of HideM, we find application overhead ranges from a 6.5% increase to a 2% reduction in runtime and observe runtime memory overhead ranging from 0.04% to 25%. HideM requires adversaries to guess ROP gadget locations making exploitation unreliable. We find adversaries have less than a 16% chance of correctly guessing a single gadget across all 28 evaluated applications. Thus, HideM is a practical system for protecting vulnerable applications which leak executable data.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*; D.4.6 [**Operating System**]: Security and Protection—*Information Flow Controls; Access Controls*

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

CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.

http://dx.doi.org/10.1145/2699026.2699107

General Terms

Security

Keywords

return-oriented programming; information leaks; memory disclosure exploits; code reuse attacks; memory protection

1. INTRODUCTION

Protecting memory is a critical component to ensuring the security of a system. Process memory contains many types of sensitive information including code, keys, and other secrets. Contemporary computer hardware contains page level memory protections preventing reads and writes. These protections provide isolation between address spaces (e.g. user and kernel-space).

Recent processor extensions seek to protect memory within an address space. For example, the no-execute (NX) protection bit prevents execution of specific memory pages (e.g., stacks). Unfortunately, techniques such as Return Oriented Programming (ROP) allow successful exploitation without executing these memory pages [23].

A common requirement for modern exploits to bypass system protections (e.g., ASLR, DEP) is reading memory. Specifically, a memory disclosure vulnerability is used to disclose code locations and values to ensure exploit reliability and correctness [17]. Memory disclosure vulnerabilities that leak code instructions are fundamentally possible because *execute permission always implies read permission* on commodity hardware. As a result, advanced protections such as fine-grained ASLR [14, 34] and Control Flow Integrity [2, 37] can be bypassed leading to exploitation [24, 10].

Execute-only memory is a well defined and understood technique for protecting the contents of memory. Multics, a classical secure system design and architecture, included an execute-only bit for memory pages [7]. XoM implemented execute-only memory by encrypting executable data and only decrypting on instruction loads. XoM required a custom processor architecture and suffered from poor performance making adoption difficult. Regardless, both approaches are too coarse grained in their protections.

We seek to broadly protect critical contents of userspace memory by leveraging a concept we call **code hiding**. Code hiding is inspired by PaX [28] for enabling no-execute memory without hardware extensions, as well as by advanced rootkit hiding that prevents forensic analysis [25]. We propose code hiding to protect userspace memory from being read by a malicious or vulnerable process. Code hiding is

enabled using the unique features of the split Translation Lookaside Buffer (TLB) architecture to configure reads of executable pages on contemporary commodity hardware.

In this paper we propose HideM, a system that disallows userspace code from arbitrarily reading critical data in its own memory address space. HideM is built on code hiding to prevent reading executable data of existing legacy and COTS binaries. We observe the majority of code does not need to be read but only executed. We propose *code reading* policy as an approach to enforce fine grained read access on executable pages. Applications can also leverage HideM to protect sensitive data by encoding critical data in executable pages. The design of HideM allows integration with existing advanced security techniques (e.g., fine-grained ASLR [34, 14] and CFI approaches [37]). HideM ensures memory disclosure vulnerabilities cannot be used to find ROP gadgets to enable reliable exploitation, discover vulnerabilities in memory, and enable some forms of data leakage. Furthermore, HideM is generic and can be enabled on existing commercial hardware with minimal performance overhead to protect memory against disclosures.

Code reading policies are created based on data that may be legitimately read by code. For example, the GNU GCC compiler will embed exception handling data in code pages as an *.eh_frame* section. Policies are generated based on binary structure (e.g., read-only data in executable pages) and code/function symbols. To identify what code needs to be read as data, we perform binary analysis to recover code symbols from executable sections and identify data in code. We perform minimal manual analysis to verify the identification of data in code as correctly identifying all data in code is provably undecidable [35]

HideM uses *code reading* policies to divide read data from **executable data** (e.g., machine code) on executable pages. Shadow memory pages are created containing only required readable data or executable data. The OS kernel configures the hardware split-TLB to hide executable data from userspace read access. As a result, HideM can transparently apply *code reading* policies to commercial off-the-shelf (COTS) binaries.

In this paper, we make the following contributions:

- We design and implement HideM for protecting against the broader threat of information disclosure of process memory. HideM leverages *code hiding* as a new security mechanism to provide fine-grained userspace read access without changing current commodity hardware.
- We propose *code reading* policy to automatically configure userspace reads of binary executable pages. Policy is enforced at runtime and protects executable data vulnerable to memory disclosure.
- We evaluate compatibility, performance, and security impact of HideM. We find that HideM has limited impact on performance. The runtime increase ranges from 6.5% to a 2% reduction. We observe working memory increases ranging from 0.04% to 25%. Furthermore, HideM raises the level of security for protected binaries by reducing the probability of reliable exploit. We find an adversary has a less than 16% chance of guessing a single valid ROP gadget.

The remainder of this paper is as follows. Section 2 has a background on memory access and memory disclosures. Section 3 provides an overview, Section 4 discusses design, and Section 5 provides implementation details. Section 6 evaluates HideM. Section 7 discusses limitations and future work. Section 8 has related work. Section 9 concludes.

2. BACKGROUND AND MOTIVATION

We briefly provide background and motivation for HideM. First, we discuss memory accesses using hardware caching of virtual to physical mappings. Then, we motivate HideM looking at the threat of memory disclosure on existing security protections.

2.1 Memory Access via TLB

The Translation Lookaside Buffer (TLB) is a cache used by processors to reduce the cost of continuous memory accesses. The TLB stores mappings from page numbers (i.e., upper bits of a linear/virtual address) to physical frame (i.e., upper bits of physical address) along with page status and permissions. The TLB obviates the need for the system Memory Management Unit to walk page-tables for each memory access. The TLB also assists in MMU translations (i.e., walking pages-tables) by adding entries for intermediate page-table values (i.e., top level page directories).

Operations of TLB: TLB operations for managing entries are architecture specific. For example, SPARC implements TLB in software and thus the OS manages the TLB by adding and evicting entries [36]. On the other hand, x86 and ARM architectures use both hardware and software for TLB management [22, 13]. Specifically, entries are only added by hardware after the MMU walks page-tables. Entries are *flushed* (i.e., evicted) by both hardware events such as task-switch or using privileged CPU instructions.

Split-TLB Architecture: Processors commonly contain two TLBs, a DTLB to handle data accesses and an ITLB to handle instruction fetches. This split architecture allows for better locality of accessed memory [18]. In normal execution, the DTLB and ITLB are synchronized and contain the same values for a given address. TLB flushes of a specific address will remove associated entries from both the ITLB and DTLB. However, entries are added based on the type of CPU access. If an instruction is executed on a page, an ITLB entry is added. If memory is read, for example via a *mov* instruction, a DTLB entry is added.

2.2 Memory Disclosure

Memory disclosures are a subset of information leakage vulnerabilities in which an adversary gains unauthorized access to read raw memory. An adversary can then leak sensitive information such as encryption keys, passwords, or executable data. Using leaked executable code and associated addresses, an adversary can build reliable ROP based exploits in an automatic and just-in-time fashion [24]. Memory disclosures have also been used in bypassing CFI enforcement. Goktas et al. showed relaxed CFI code transfer enforcement can lead to ROP based exploitation [10]. The authors' exploit relied on a memory disclosure vulnerability to find function call and return stubs for trampolines.

There is a growing need to protect systems against memory disclosures; however existing research is limited. In independent work, Backes et al. [4] seek to provide "Execute Not Read" (XnR) permissions on code pages. Unfortunately, the proposed approach allows reading of the currently executing

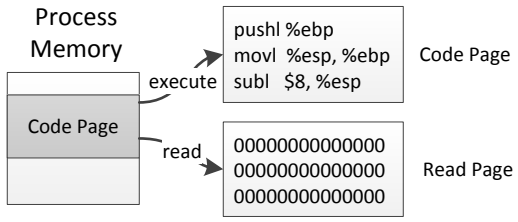

Figure 1: High level view of memory hiding shows different memory values based on memory access type (i.e., read or execute).

code page (and optionally surrounding pages within a window). These pages can be identified by return addresses on the stack and therefore can used in a JITROP attack [24]. Furthermore, their approach does not address allowing legitimate code reads present in PIE and C++ binaries. Our proposed approach provides finer-grained protection to allow legitimate reads of code page.

3. OVERVIEW

The goal of HideM is to protect binaries from leaking executable data from arbitrary reads by userspace code. In turn, HideM thwarts memory disclosure vulnerabilities and JIT-ROP style attacks described in previous work [24, 17, 10]. At a high level, HideM simply displays different contents of memory for different CPU operations. Figure 1 depicts the basic protection of HideM wherein a single memory address provides different content based on read or execute operations. Specifically, read operations will access **read pages** and instruction fetches will access **code pages**. As a result, memory containing executable data that is *hidden* cannot be read by processes or other traditional memory reading operations. Similarly, hidden read-only data (e.g., jump tables, binary headers) on executable pages can no longer be used for execution.

The properties of HideM harden the security of existing systems and provide new opportunities to protect data in memory. However, enabling seamless memory protections provided by HideM faces challenges:

C-1: *Ability to execute memory implicitly allows read access.* Existing commodity hardware architectures such as x86 and ARM do not provide distinctions between read and execute permission. Executable memory implicitly contains read permissions in these architectures. As a result, all code can be read as data. Memory hiding must differentiate memory accesses into read and execute operations.

C-2: *Executable pages of binaries often contain read-only data that is read during execution.* For example, GNU GCC references immediate values in code as memory addresses of functions. We must ensure that we allow legitimate reads of such data for correct execution. This requires fine-grained read-access at sub-page granularity.

C-3: *Support for legacy and commercial-off-the-self binaries without code symbols.* We must determine what data needs to be read by code to build code reading policies. Symbols can be added at compile time with access to source code but often removed in final binaries.

To overcome the above challenges, we use a combination of binary analysis (**C-3**), OS kernel memory management (**C-2**), and hardware memory features (**C-1**). Specifically, HideM performs lightweight data analysis to identify read data (e.g. jump-tables, or embedded data) and symbols for data residing in executable pages (**C-3**). At binary load time, HideM generates a read policy based on identified symbols and in-memory binary layout to allow code to read legitimate data from memory (**C-2**). The policy is then enforced on memory access through OS kernel hardware configuration using the memory address caching of TLBs (**C-1**).

We identify that code pages often contain read-only data and function symbols that are read during legitimate execution (**C-2**). Thus, memory hiding must allow for selective reads of data on code pages. HideM generates a read policy that allows for such data to be legitimately read by code. The policy is generated using binary and symbol information to create a shadow page that contains necessary data for correct execution. The policy is enforced by adding a page of read-only data to the DTLB and a page of instructions to the ITLB. When the DTLB or ITLB is not configured, all page accesses are trapped allowing TLB reconfiguration.

The read policy is generated based on binary section information along with symbols and read-only data in executable sections. Symbols are often found as part of relocation information. Unfortunately, many compilers such as GNU GCC and LLVM remove relocation information by default and thus COTS binary will not have symbols for code sections (**C-3**). We reconstruct partial relocation information to identify symbols in code sections using static binary data analysis. Specifically, we log the locations of all possible immediate values in code sections that may point to code addresses. These symbols represent the expected data that may be read by code during execution. In a small number of cases, read-only data such as jump-tables are embedded in executable data. We adapt algorithms and heuristics based on previous research [37] to identify subject data in code. After which, we manually verify data identified as ambiguous because distinguishing data from code is a proven undecidable problem [35]. Fortunately, our experience using HideM indicated that ambiguous data only occurred in a few libraries. We also use binary layout information to identify compiler marked read-only data in code pages.

The general flow of HideM is shown in Figure 2. Prior to execution, binaries are analyzed and data locations in executable sections are added to the binary as non-loaded data. Upon loading of the binary, the OS will identify the binary as protected by HideM, extract the data locations, and identify load information (i.e., segment permissions, section permission, offsets). Using this information, HideM builds a read policy containing executable data regions and data required to be read. On first load of a protected page into memory (e.g., on page-fault), the policy is applied to the faulting page by identifying the locations of read-only data, executable data, and data locations on the page. The policy is then enforced through hardware configurations set by the OS. Specifically, we leverage the concepts of desynchronizing the split ITLB/DTLB to seamlessly enforce different memory permissions for different CPU operations (i.e., instruction loads and memory reads) (**C-1**).

Assumptions: We assume that the system has a Memory Management Unit (MMU) configured with virtual addressing and page-tables. This requirement effectively enables

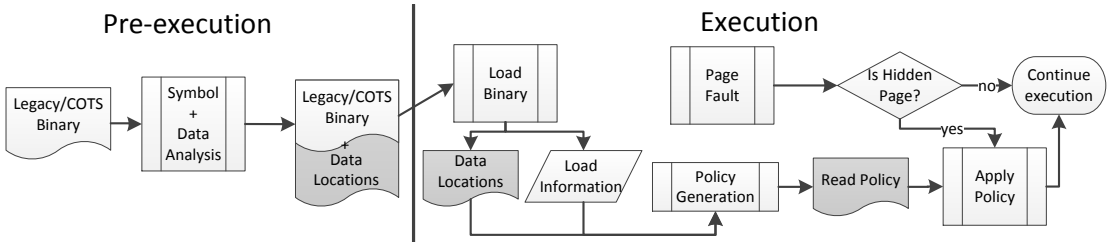

Figure 2: HideM overview: Prior to execution, binaries are analyzed to identify data read by code. Locations are used to create *code reading* policy. Access is enforced through page permissions and de-synchronized TLB.

TLB caching and virtual address permissions (e.g., supervisory, non-writable, non-execute, non-present). We assume that all code pages for hiding are non-writable. Our approach assumes the target hardware has a split-TLB architecture and does not have a unified TLB cache. We discuss implications of unified TLB caches in Section 7.

Threat Model: We trust that the operating system kernel is not modified or compromised. Attackers have access to memory disclosure vulnerabilities that can read arbitrary userspace virtual memory of a vulnerable process. However, the attackers do not have prior knowledge about known locations of ROP gadgets in memory. Specifically, the attacker cannot assume ROP gadget locations based on binary load location.

4. HIDEM

Next, we discuss the details of HideM. In Section 4.1, we discuss enforcing memory permissions based on memory access types using TLB de-synchronization and OS kernel page table management. In Section 4.2, we discuss generating and enforcing a code reading policy. Section 4.3 discusses HideM support for protecting userspace executable memory allocations.

4.1 Memory Permissions Enforcement

HideM provides separate enforcement for reading and executing data in memory for commodity systems. Memory permissions are enforced by (1) trapping all accesses to hidden memory (i.e., protected executable pages) by default, and (2) configuring the ITLB and DTLB to allow access to distinct physical pages for code and data respectively. Once the TLB is configured, future memory accesses of the page will not trap, allowing execution to continue. The TLB needs reconfiguration after the TLB is cleared. Figure 3 shows an overview of enforcing memory protections. The hardware traps hidden memory access based on virtual memory page permissions. The page-fault handler is then responsible for *priming* the TLB with correct virtual to physical page mappings and permissions.

4.1.1 Permissions Management

By default, all accesses to hidden memory must be trapped when the TLB is not configured. This ensures that the OS kernel can configure memory permissions for hidden memory prior to access. This is similar to standard page-faults whereby the OS kernel is responsible for correctly configuring page-table entries. However, the lack of distinction between execute and read permissions in page-table structures limits the ability to handle hidden memory in hardware. Instead, the OS must intervene on access to configure permissions accordingly.

HideM uses page-faults as a gate to mediate all accesses to hidden memory. To enable page-faults by default, the *user* bit is cleared for page-table entries of protected memory effectively marking the memory for *supervisor* access only. This bit is commonly cleared for kernel pages and is supported on all contemporary processors. As a result, all accesses from userspace to the protected page will cause a page-fault. Note that all HideM page-table entries by default have a physical address of a shared zero page. This prevents any userspace controlled memory from being used for privilege escalation.

Blocking all userspace accesses to hidden memory will not allow for useful execution. To allow execution to continue, the page-tables are temporarily configured with permissions that allow memory access based on CPU operation of the fault. For example, on an instruction fetch fault, HideM temporarily maps the page-table entry to the physical address of the code page and sets permission for userspace access. Next, the TLBs are de-synchronized to allow execution, but block future reads. After de-synchronization, the page-table entry that blocks access by default is restored.

4.1.2 TLB Priming and De-synchronization

The split-TLB architecture stores separate virtual address to physical address mappings along with permissions for instruction loads and data reads. In traditional system operations, page-table configurations are added to TLB caches upon memory access and instruction fetches. As a result, when data from a code page is both read and executed, the ITLB and DTLB values will be identical and thus synchronized.

TLB de-synchronization occurs when the ITLB and DTLB contain different entries for the same virtual address. The implication of de-synchronization allows a single virtual address access to result in disparate physical page accesses. For example, Figure 3 depicts a de-synchronization where an ITLB entry for virtual address 0x40000 maps to physical address 0x1000 and the DTLB has a mapping for virtual address 0x40000 to physical address 0x4000. Unfortunately, TLB operations for adding entries to the ITLB and DTLB are limited to specific architectures as discussed in Section 2.1. As a result, we must provide a way to add entries to the ITLB and DTLB thereby enabling TLB de-synchronization.

TLB priming is the process of configuring hardware to add entries to either the ITLB or DTLB. The idea of TLB priming uses the knowledge of how hardware adds entries to respective caches. In the case of the ITLB, an entry is added

after the page-table is successfully walked by the configured MMU to fetch an instruction. Similarly, a DTLB entry is added upon walking the page-table for a read operation. A successful page-table walk occurs when the physical page for a page-table entry is present and contains the correct permissions for the operation.

Priming of the TLB begins when a page-fault occurs on access to hidden memory. The TLB entry for the address is flushed for the virtual address accessed by the operation. This is required to ensure there are no existing entries that may contain wrong permissions (e.g., supervisory). Two page-table entries are generated to allow userspace access. One entry pertains to a read operation and the other to an execute operation and thus the corresponding entries are configured with the physical page addresses accordingly.

The page-tables of the faulted process are briefly updated with the read page entry and the code page entry. When the page-table is set with the read entry, the DTLB can be primed by reading one byte from the page using the faulted virtual address. Similarly, the ITLB is primed after the code page is mapped in the page-tables and then executing an instruction in the page. On the first page-fault for a hidden page, the executable data page is mapped to kernel space and the page is searched to find a *return gadget* or similar instruction (e.g., ret, jmp %reg). Searching for the gadget using kernelspace addressing prevents priming the DTLB with a userspace address to code page entry. HideM then calls into the page at the gadget location using the userspace virtual address of the return gadget. After priming, the page-table is updated to the original faulting value. For processors that support SMEP and SMAP, protections are disabled immediately before priming and enabled after priming.

HideM primes both the ITLB and DTLB on each page-fault. This prevents multiple page-faults from occurring by code that reads data on the executable pages.

4.1.3 Non-Execute Optimization

For processors supporting non-execute (NX) page permissions, HideM can use the NX bit to trap access instead of the *user* bit. By default, HideM protected pages will all be non-execute and mapped to the read page allowing applications to read the data but not execute. Page-faults for priming only occur on execution. This is beneficial in applications containing large amounts of read-only data on executable pages.

4.2 Code Reading Policy

De-synchronized TLBs allow page-level read and execute policies to be set on memory. For example, HideM can enforce execute-only memory wherein executable pages cannot be read. Unfortunately, executable pages of existing binaries often contain data that is read during execution. As a result, execute-only pages are too broad for applying permissions to binaries. Thus, HideM must allow selective reads of executable pages to enable correct execution.

Code reading policy identifies the bytes on an executable page that may be read as data. Executable pages read as data will only contain required data for execution and not the majority of code.

Data required to be read on executable pages can be categorized into two types:

DT-1 read-only data on executable pages (e.g., jump tables, exception handler data, binary headers)

DT-2 immediate values used for pointer arithmetic and function calls (e.g., symbol addresses, call offsets).

HideM instances of **DT-1** and **DT-2** data are represented as byte ranges and offsets in executable pages. After all of the data embed in executable pages is identified, the offsets and ranges of identified data are added to the binary in a new section which is processed during binary load time. Next, we provide details for identifying **DT-1** and **DT-2** data using binary structure and binary analysis.

4.2.1 Identifying DT-1 Data

DT-1 data can (1) reside outside executable data (e.g., binary headers) or (2) be embedded with executable data (e.g., jump-tables). This data is never executed by a process and thus can be removed from code pages used for execution.

Read-only **DT-1** data outside of executable sections can be identified using binary structure and metadata. For example, ELF [6] binaries contain *sections* which identifies contents and permissions. Sections are associated with *segments* which defines in-memory binary locations and permissions. Section permissions are disjoint from segments allowing read-only sections to be part of an executable segment. GNU GCC default linker scripts combine read-only sections (e.g., .rodata, .eh_frame_hdr, .eh_frame, .rela.dyn) with executable sections (e.g., .init, .plt, .text, .fini) into one segment and thus the same physical pages of memory. PE/COFF [31] read-only data is by default configured in different pages simplifying policy generation on Windows.

Read-only **DT-1** data embedded within executable sections is identified using binary analysis. We apply a modified algorithm and heuristic presented by Zhang and Sekar [37] to discover jump-tables and other non-standard data in code. Specifically, we run the algorithm to first identify all control flow locations and build a control flow graph. The algorithm identifies jump-tables to determine all indirect control flow locations required for enforcing control flow integrity. To identify data that is not noted as jump tables, regions of the binary not belonging to the generated call graph are marked as unknown. These regions map to gaps and errors in disassembly. Complete and correct identification of data in code requires correct disassembly, which is a known undecidable problem [35]. Thus, we perform minimal manual analysis of unknown regions to verify contents as data. Due to space constraints, we only briefly describe the algorithm for disassembly and analysis. We encourage the reader to read previous work [37] for a more detailed discussion of the algorithm.

4.2.2 Identifying DT-2 Data

DT-2 data is often encoded by linkers as relocation information. Relocation data is commonly found in PE/COFF binaries and used for enabling binary relocation and ASLR. Thus, Windows binaries with ASLR enabled do not require analysis to identify **DT-2** data. However, ELF binaries often have relocation information stripped which makes generating a code reading policy from COTS binaries less feasible.

We look to recover relocation information by analyzing binaries to identify potential symbol references. Unfortunately, complete reconstruction of relocation information from binaries is an unsolved problem [20]. However, we identify that reconstructing **DT-2** data is a confined prob-

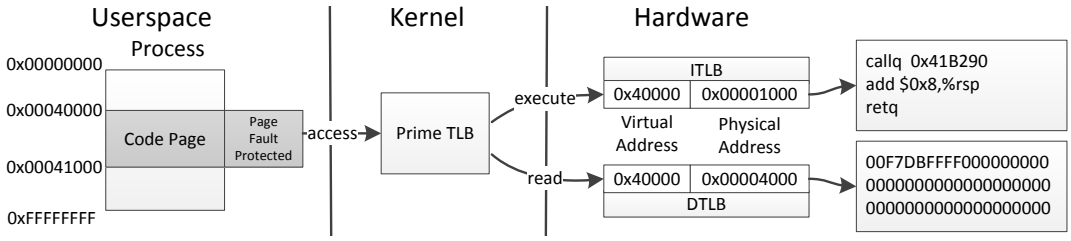

Figure 3: Enforcing HideM protections: Page-faults trap all accesses to hidden pages. Then HideM configures hardware to differentiate reads and fetches. Access is allowed while TLB entries are not evicted.

lem with a number of assumptions and restrictions. As a result, **DT-2** data can be reconstructed from static binary analysis.

DT-2 data assumes that memory being read as data will reside in executable sections. This data is categorized into two groups: (1) immediate values that represent addresses located within an executable section, and (2) instruction pointer relative offsets resolving to an executable section.

Prior to execution, **DT-2** data is identified using binary headers and analysis of the disassembled binary. Executable sections and their ranges in virtual memory are found using binary headers and the base load address. The binary is disassembled and searched for operations containing immediate values and instruction relative pointer operations that resolve to addresses located in code sections. For example, a x86 *call* instruction will specify a relative address to execute. The offsets used as part of the instruction must be readable. Identifying **DT-2** data is performed during analysis to identify **DT-1** data embeded in code.

4.2.3 Applying the Policy

HideM uses **DT-1** and **DT-2** data to divide executable pages into read pages (i.e., containing only **DT-1** and **DT-2** data) and code pages (i.e., executable data which includes **DT-2** data). Once divided, these pages are used in TLB priming and de-synchronization. TLB hardware then enforces the read and execute policies on these pages as described in Section 4.1.2.

At binary load time, **DT-1** and **DT-2** data locations are recorded and associated with virtual memory address allocations (e.g., Virtual Memory Areas for Linux, Virtual Address Descriptors for Windows). The associated values are then used to generate separate code-only and read-only pages when an executable page is mapped into memory.

On a page-fault of a non-present executable page, **DT-1** and **DT-2** ranges are checked to determine if the executable page has data that can be read by code. If the faulting page does not have **DT-1** and **DT-2** ranges, the code page should never be read as code and thus the DTLB is only primed with a shared zeroed page. If the executable page contains **DT-1** or **DT-2** ranges, then a read page is created and the code page is modified with respect to **DT-1** and **DT-2** ranges. Specifically, the **DT-1** and **DT-2** ranges from the code page are copied to the read page. The code page has **DT-1** ranges filled with halt instructions. If the page contains only **DT-1** data, then the page is not modified and a shared code-page filled with halt instructions is used to prime the ITLB. By isolating **DT-1** data and executable data into separate pages, the policy prevents read-only data on executable pages from being used as ROP gadgets.

4.2.4 Hardening HideM against ROP Exploits

DT-2 data must be both readable and executable. Therefore, the same **DT-2** data values will appear in both the read and the code shadow pages. This property allows an adversary to potentially build ROP gadgets from **DT-2** data. Since **DT-2** data is limited in size (4 bytes) and is non-contiguous, the complexity of potential ROP gadgets is limited. However, HideM strives to make the discovery of **DT-2** ROP gadgets as difficult as possible.

As depicted in Figure 3, discovering **DT-2** data in read pages is trivial: all non-**DT-2** data is zero. We harden HideM by replacing the zero data with random data. The random data is created to match the form of **DT-2** data. Therefore the adversary must guess if a memory location value is **DT-2** data (i.e., executable) or not. An incorrect guess will cause the program to crash.

HideM generates random, fake **DT-2** data based on the types of values that are stored in **DT-2** data. Specifically, **DT-2** data is instruction pointer relative addresses or values. We fill read pages with false data that is relative to both the location being filed and the executable sections for the binary loaded at this location. We also randomly align this data in the page to prevent guessing **DT-2** data locations based on alignment.

4.3 Protecting Userspace Allocated Executable Memory

HideM generates code reading policy based on **DT-1** and **DT-2** ranges from the kernel at process load time. However, many applications load shared libraries from userspace (i.e., dynamic linking) or dynamically allocate executable memory at runtime (e.g., scripting). This presents a challenge, since HideM does not have the context required generate code reading policy for these executable pages.

To address this challenge, HideM provides an interface for userspace applications to provide context required for protecting runtime allocated memory. Memory may be flagged as HideM protected at allocation time or by changing protections. In effect, this makes protected memory execute-only. The userspace application can then provide **DT-1** and **DT-2** ranges to generate the shadow pages for priming. Note that, HideM only enforces protection after the memory has been made non-writable. As a result, HideM can protect dynamically loaded libraries and other dynamically allocated memory at runtime.

5. IMPLEMENTATION

Our HideM implementation uses a modified tool from previous research [37] to disassemble and parse binaries to identify **DT-1** and **DT-2** data. Specifically, we added support

for 64-bit binaries. **DT-1** and **DT-2** data locations are then written back into the binary as a separate non-loaded section. HideM protected binaries have their in-memory permissions of executable data modified with a new protection flag. Note that HideM converted binaries also work on legacy systems.

We implemented HideM for Linux Kernel 3.10.12 with 64-bit ELF support. Currently, we do not have 32-bit support, however this is an engineering issue. We augment ELF binary loading in the kernel to extract **DT-1** and **DT-2** ranges from the HideM enabled binaries. We also modify interpreter loading to protect HideM enabled dynamic loaders or interpreters as discussed in Section 4.3. Code reading policy is associated with binary load addresses and the provided **DT-1** and **DT-2** ranges. False executable data used for hardening against ROP exploits is generated using a hardware random number generator.

To test support for shared libraries, we modified the LD-loader for GLIB 2.18. The LD-loader tells the kernel which binary and data locations to protect. In total we added 53 lines to enable HideM support of GLIB. We believe minimal effort would be required for porting other dynamic loaders such as Apache's mod_so for Dynamic Shared Objects.

6. EMPIRICAL EVALUATION

We look to evaluate the practicality of adopting HideM to existing platforms by evaluating three different aspects of HideM. First, we evaluate HideM's impact on system performance and resources. Next, we evaluate HideM's compatibility with COTS applications. Finally, we provide a security evaluation to investigate the reduction of valid gadgets and the probability that an adversary can build a valid ROP exploit for a HideM protected binary.

6.1 Experimental Setup

We performed our evaluation on an IBM LS22 blade server with two Quad-Core AMD Opteron 2384 processors with 32GB of RAM running 64-bit Ubuntu 12.04.4 LTS. Each core can store 1024 4KB page entries. This processor does not support SMEP or SMAP and thus are not considered in the performance evaluation. However, disabling and enabling SMEP and SMAP consists of twelve instructions in total.

We acquired 28 different applications for evaluating HideM. These applications were chosen based on the dataset provided in previous work [37]. 19 applications are part of SpecCPU2006 and 9 applications are non-trivial common Linux user applications. We built the SpecCPU2006 benchmark applications with a modified base AMD64 configuration adding only "check_md5=0" to prevent rebuilding applications that were converted to HideM. The complete list of applications can be viewed in Figure 5.

We converted the 28 applications along with all required shared libraries to enable HideM protection. In total, we converted 442 binaries totaling 441MB. We identified that 3% (13 binaries) of 442 binaries contained data embedded in code of which only *libcrypto.so* required manual analysis for unknown data regions containing cryptographic algorithm data.

Each binary is converted by identifying **DT-1** and **DT-2** ranges in executable pages and writing the information back into the binary as a separate ELF section. We first converted the HideM supported GLIB ld-loader and shared libraries. We then converted all binaries required for running the 28 applications. To identify all required shared libraries, we ran all 28 applications with LD_DEBUG enabled for printing all files opened by the loader. This provided a list of all loaded binaries along with respective locations. We found that *ldd* did not provide a list of all binaries loaded into memory. The 28 application binaries had their ELF interpreter modified to load the HideM enabled GLIB ld-loader.

6.2 Performance Evaluation

HideM augments execution to enforce code reading policy. In turn, HideM may impact the performance of a running application when enforcing a de-synchronized TLB configuration. HideM also requires additional runtime memory for generated shadow pages. Finally, converting binaries to HideM support requires additional diskspace. Next, we evaluate HideM's impact of each of these categories.

6.2.1 Runtime Overhead

To investigate HideM's impact on runtime performance, we observed the runtime overhead of the 19 converted SpecCPU2006 applications. The benchmark was run with the command line options "–size=ref -noreportable –action=run –nobuild". This configuration constitutes a reportable run with the number of run iterations set to three for each application. However, we are required to set the "–noreportable" switch to force execution of the benchmark when "–nobuild" and "check_md5=0" are set.

Figure 4 shows the percent overhead for each application as reported by SpecCPU2006. The percent overhead observed ranged from a 6.5% increase in runtime to a 2% decrease in runtime with an average increase of 1.49% and the median increase of 0.51%.

Many applications have minimal runtime overhead and even a performance increase. For example, *h264ref* has a 2% reduction in runtime. Low performance overhead and increases can occur for three reasons. First, read policy generation is lazy only occurring on first access to a protected page. Thus, the cost of generating a shadow page is only incurred after a page is accessed. The TLBs are then primed on first page-fault. Second, TLB priming opportunistically caches page mappings for both data and instructions on a single fault. Intermediate page level entries are cached preventing the need to walk the entire page-table. Standard hardware page-table walks may require caching all intermediate values. Finally, execution paths that do not cross many pages may never fill the TLB cache and thus may never be re-primed. Applications exhibiting noticeable overhead such as *perlbench*, *soplex*, and *hmmer* have TLB entries evicted requiring re-priming.

6.2.2 Runtime Memory Overhead

HideM requires extra runtime memory for storing code reading policy and generated shadow pages. Shadow pages are generated under two conditions: (1) **DT-1** data and executable instructions reside on the same page or (2) **DT-2** data resides on an executable page. In the worst case, HideM will consume approximately twice the amount of executable pages allocated for a given binary.

We recorded the maximum Resident Set Size (RSS) for each tested application during our compatibility testing discussed in Section 6.3. We then ran each application without HideM protections under the same conditions (i.e., same

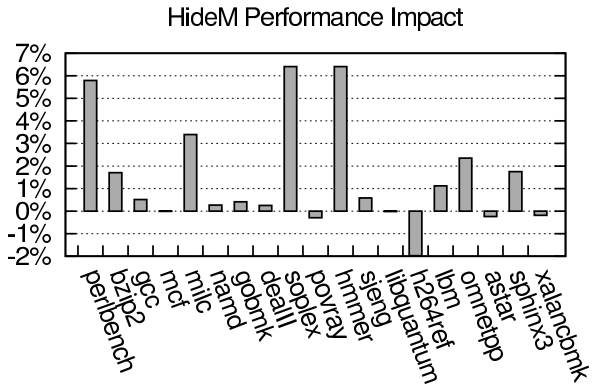

Figure 4: Percent execution overhead for SpecCpu2006 benchmarked applications.

kernel and modified GLIB) and workload recording the maximum observed RSS. Figure 5 shows the percent maximum RSS memory overhead for each of the 28 applications. The observed percent overhead increase ranges from 0.04% to 25% with an average of 4%. Applications exhibiting small amounts of overhead allocate significant amounts of data and or have tight execution flows which does not generate a significant number of shadow pages. We looked at the amount of memory increase in MBs for the five applications with the largest percent increase: smplayer/24.41MB, dumpcap/2.89MB, lyx/24.03MB, povray/3MB, lynx/4.09MB. The large memory footprint of smplayer and lyx occurs because execution covered a significant number of code pages across the loaded binaries.

6.2.3 Disk Overhead

To enable HideM support, binaries must provide **DT-1** and **DT-2** ranges residing in executable pages. This data is added to binaries during the conversion process and thus requires extra disk storage. We report the disk storage costs for enabling HideM support of the converted binaries.

We calculated the percent difference in on-disk binary sizes for all 443 converted binaries. We observed the average amount of data required to enable HideM support was 4.9% and medium of 4.2%. The overhead of the 28 application range 1% to 11.8%. Figure 5 shows the disk storage overhead for each of the 28 application binaries. HideM has the potential to leverage relocation information when available thereby reducing this cost.

6.3 Compatibility

HideM makes modest changes to binary loading and execution but should not interfere with correct execution. To evaluate HideM's correctness and compatibility with existing COTS applications, we ran the 9 non-trivial applications under application specific workloads. The list of applications tested along with workloads can be found in Table 1.

We also ran the 19 SpecCPU2006 benchmark applications with the same configuration discussed in Section 6.2.1. SpecCPU2006 verified the expected output of each benchmark run and did not report any errors.

6.4 Security Evaluation

Our goal is to determine the security advantage of protecting binaries using HideM. The main way to exploit HideM is to build ROP exploits using only gadgets found in **DT-2** data. These gadgets reside on both read pages and code pages. Gadgets found completely within **DT-2** rages are *valid* and can be used in an ROP exploit. In an unprotected binary, all identified gadgets are valid. HideM obfuscates **DT-2** data by randomizing readable data not part of **DT-1** or **DT-2** ranges with values that mimic **DT-2** data. Gadgets identified as part of this data are non-valid.

Table 1: Non-trivial applications evaluated with specific workloads to test compatibility with HideM.

Binary Name	# Experiment
wireshark v1.6.7	Captured packets for 10 minutes, filtered TCP on port 80
dumpcap v1.6.7	Captured packets for 10 minutes
gimp v2.6.12	Open JPEG, blur, enhanced, cropped
gedit v3.4.1	Opened 189KB file, copied, pasted, saved
lynx v2.8.8	Opened ncsu.edu website, navigated, posted forms in search
python v2.7	Ran Volatility 2.3.1 pslist command on 1.0GB memory dump
emacs v23.3.1	Opened, edited, saved text files of size 200KB and 1MB
lyx v2.0.2	Opened classic thesis template, made modifications, compiled
smplayer v0.7.0	Played 10 minute 720p video

HideM provides security advantages in two ways. First, HideM reduces the number of valid gadgets that can be used to build exploits; thus, making exploit synthesis more difficult. Second, HideM introduces non-valid gadgets into readable pages. As a result, an adversary must guess valid unique gadgets and locations within **DT-2** to enable successful exploit generation.

First, we identified all valid and non-valid gadgets in memory for both HideM protected binaries and non-protected binaries for all 28 tested applications. To identify gadgets we used two open source ROP gadget finder tools: ROPGadget v4.0.4 [19] and RP++ v0.4 [1]. We chose these utilities for their flexibility, support of 64-bit architectures, and their difference in types of gadgets reported. ROPGadget reports a limited set of expressive gadgets (e.g., pop %reg ret) while RP++ is more aggressive providing many more *less expressive* gadgets. These tools do not enable synthesizing available gadgets to build exploits. Tools such as Q [21] do synthesize gadgets to build exploits but do not support 64-bit architectures. We dumped all reported gadgets from the two utilities and identified valid and unique sets of gadgets. Table 2 contains the raw gadget data for all 28 application binaries, which will be used for calculating the probability of exploitation. Gadget sizes searched for HideM protected binaries were limited to a length of 4-bytes or less. This insight comes from the observation that symbols composing **DT-2** data are likely 4-bytes for 64-bit binaries and are never contiguous. We use this data going forward to assess the security advantages of HideM.

Reduction in Valid Gadgets: Without hardening HideM, **DT-2** data stands out in memory. For example, Figure 3 shows a read page with a single valid **DT-2** value. HideM reduces the set of valid gadgets available for exploitation. The set of valid unique gadgets in a HideM protected binary is a subset of all valid unique gadgets for a non-protected binary. Thus an adversary is more restricted when synthesizing gadgets to build an exploit.

The total reduction in valid gadgets can be seen in Table 2 by comparing the "Orig." unique gadgets to "Valid HideM"

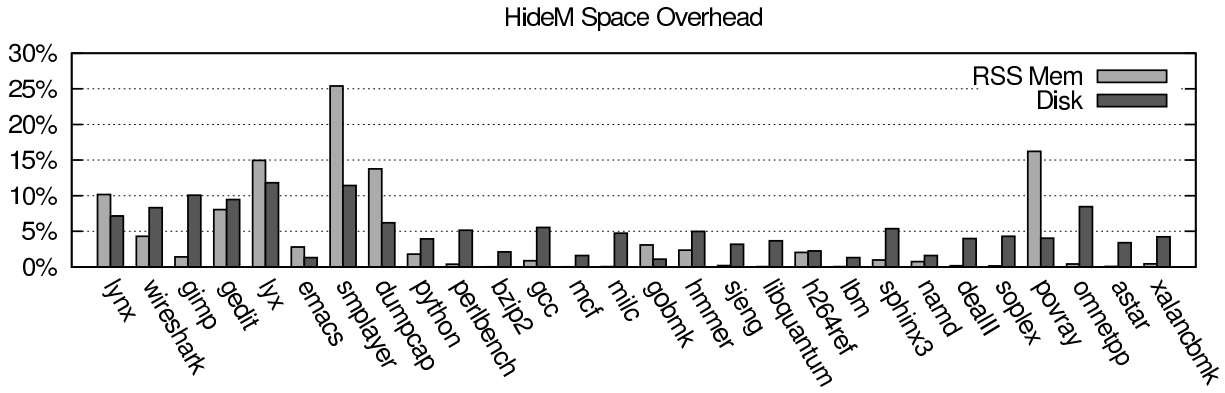

Figure 5: Percent memory and disk overhead observed for each application.

unique gadgets. We found HideM reduces the number of valid unique gadgets identified by ROPGadget ranging from 38.24-100% and for RP++ ranging from 99.3-99.92%. The average reduction for ROPGadget is 77.33%. In general, the reduction is lower for applications with a larger number of **DT-2** ranges, which leads to more potential gadgets.

Probability of Exploitation: Hardening HideM adds false data to read pages and forces an adversary to guess the locations of valid gadgets. As a result, an adversary cannot simply identify **DT-2** data to build an exploit. Thus, adversarial exploitation of HideM protected binaries is based on the probability of correctly choosing a series of valid gadgets.

We can reduce the problem of building a successful exploit to modeling as "ordered sampling without replacement". The adversary chooses N different gadgets from a set of unique gadgets (U_g) at different address locations to build an exploit. There is a set of unique valid gadgets (U_{vg}), which is the subset of unique gadgets U_g containing at least one valid gadget. Choosing N unique valid gadgets from U_{vg} given the set of all unique gadgets U_g constitutes a successful exploit and takes the standard form

$$\frac{{}_{U_{vg}}P_N}{{}_{U_g}P_N} = \prod_{n=1}^{N}\left(\frac{U_{vg}-n-1}{U_g-n-1}\right)$$

where ${}_xP_N = \frac{x!}{(x-N)!}$ is permutation notation.

In many cases, the same gadget is often found in multiple locations of the binary. Thus, an adversary must also choose a valid location residing in **DT-2** ranges. S_g is the number of different address locations where a chosen unique gadget exists, and S_{vg} is the number of valid locations for the specific gadget. Thus, we modify the standard form to account for Svg and Sg. The probability of successful exploitation against HideM is

$$\prod_{n=1}^{N}\left(\left(\frac{U_{vg}-n-1}{U_g-n-1}\right)\left(\frac{S_{vg}}{S_g}\right)\right)$$

Svg and Sg are gadget specific and are dependent on the frequency of duplicates.

We calculated the probability of exploitation for each of the 28 application binaries protected by HideM. Svg and Sg are gadget specific; thus, we substituted $\frac{S_{vg}}{S_g}$ with an observed average for the distribution of valid gadgets to duplicate gadgets for each observed unique valid gadget. The right column of Table 2 contains the probability when only one valid gadget is required for exploitation (e.g., N=1). In reality, exploits will require more than one gadget. Furthermore, the expressiveness of gadgets size 4 or less bytes is limited and thus it is likely that more gadgets are required for exploitation [12]. All binaries have a probability of less than 16% for an adversary to correctly choose only one valid gadget. Figure 6 plots the probability of exploit (y-axis) for the 10 most vulnerable binaries over the number of gadgets in an exploit(x-axis). When the exploit requires 5 gadgets (N=5), the probability of exploit approaches zero for all binaries.

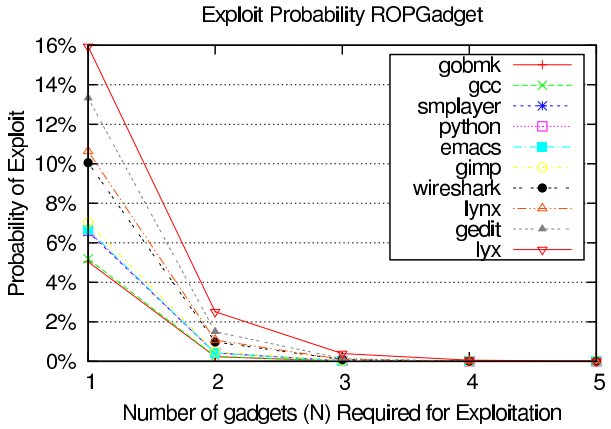

Figure 6: Probability of exploit for 10 binaries with highest probability using ROPGadget.

7. DISCUSSION

HideM does not support writable pages and thus does not support self-modifying programs. Previous work has argued that it is difficult to support self-modifying code in split memory architectures [9]. However, we believe that applying HideM with self-modifying code is possible by providing updates to code reading policy after writing to a page.

HideM uses the split-TLB architecture to differentiate memory reads from instruction fetches. Unfortunately, modern processor architectures have begun to implement a unified L2 TLB cache[1] which does not differentiate TLB entries based on the type of access. As a result, any research or technical approaches that rely on a split-TLB to differentiate memory access will not function on these processors.

[1]Unified L2 caches are standard beginning with 2008 Intel Nehalem architectures, ARM Cortex-A series, and AMD Phenom II.

Table 2: ROP gadgets statistics for ROPGadget v4.0.4 and RP++ v0.4 utilities. "Orig." contains the gadgets for the original unprotected HideM binary, "Exec" contains the gadgets in the executable region of a HideM protected binary, and "Valid" contains the gadgets that can be used for exploitation. "(A/U)" or "All/Unique" represents the number of all gadgets and unique gadgets. Probability of exploit assumes only one specific gadget is necessary for exploitation (N=1). In reality, successful exploitation will require $N > 1$.

	ROPGadget				RP++				Exploit Prob. N=1	
Binary Name	**Orig. (A/U)**	**Exec HideM (A/U)**	**Valid HideM (A/U)**	$\left(\frac{S_{vg}}{S_g}\right)$	**Orig. (A/U)**	**Exec HideM (A/U)**	**Valid HideM (A/U)**	$\left(\frac{S_{vg}}{S_g}\right)$	**ROP Gadget**	**RP++**
astar	954/31	141/20	3/3	.1698	75k/43k	88k/74k	97/44	.5963	2.55%	0.04%
bzip2	809/29	143/21	2/2	.1397	64k/39k	78k/65k	88/41	.55	1.33%	0.03%
dealII	6267/43	734/24	24/13	.0557	356k/154k	455k/372k	955/365	.5863	3.02%	0.06%
dumpcap	42/9	3/3	0/0	0.0	2372/1696	2270/1817	15/7	.7222	0.00%	0.28%
emacs	2163/31	357/27	18/11	.1637	117k/68k	233k/192k	591/204	.5765	6.67%	0.06%
gcc	3916/42	589/26	37/19	.0709	202k/100k	387k/316k	966/353	.5844	5.18%	0.07%
gedit	616/21	54/19	6/5	.5067	32k/17k	37k/31k	236/85	.7019	13.33%	0.2%
gimp	4630/34	519/21	45/17	.0869	25k/109k	353k/286k	1752/644	.7039	7.03%	0.16%
gobmk	1790/34	270/22	13/10	.111	146k/68k	171k/143k	264/100	.5142	5.04%	0.04%
h264ref	1300/35	226/20	1/1	.1429	94k/55k	141k/118k	124/50	.5348	0.71%	0.02%
hmmer	1326/34	178/20	2/1	.3333	89k/51k	117k/97k	155/62	.5774	1.67%	0.04%
lbm	806/32	127/20	1/1	.2000	63k/38k	76k/64k	63/31	.6236	1.00%	0.03%
libquantum	1122/31	183/23	3/3	.1101	72k/42k	95k/80k	97/45	.5702	1.44%	0.03%
lynx	1001/33	164/20	15/11	.1932	56k/32k	107k/89k	407/158	.671	10.63%	0.12%
lyx	7740/34	1387/28	236/21	.2124	570k/239k	789k/626k	6431/1671	.7514	15.94%	0.2%
mcf	804/30	118/22	2/2	.0734	69k/38k	73k/61k	79/35	.5722	0.67%	0.03%
milc	1010/32	154/20	7/7	.1353	72k/43k	91k/76k	105/50	.6403	4.74%	0.04%
namd	1551/32	231/20	3/3	.0972	87k/49k	141k/118k	123/63	.587	1.46%	0.03%
omnetpp	3112/42	399/23	17/10	.0937	205k/97k	236k/196k	548/205	.6088	4.08%	0.06%
perlbench	2473/41	376/23	10/5	.0948	147k/79k	230k/191k	387/144	.6210	2.06%	0.05%
povray	3254/40	409/21	8/8	.0569	160k/84k	243k/201k	292/106	.516	2.17%	0.03%
python	3464/41	286/21	20/13	.1066	177k/79k	192k/159k	459/181	.6149	6.60%	0.07%
sjeng	943/34	148/20	5/5	.1360	71k/43k	89k/74k	90/41	.5529	3.40%	0.03%
smplayer	1695/23	315/20	22/12	.1094	174k/74k	210k/172k	1154/349	.7453	6.56%	0.15%
soplex	2330/39	273/20	3/3	.0962	159k/85k	180k/150k	284/121	.5809	1.44%	0.05%
sphinx	1363/37	169/20	3/3	.1339	85k/49k	113k/94k	139/66	.6045	2.01%	0.04%
wireshark	1170/25	197/22	20/14	.1579	67k/37k	128k/106k	655/257	.7011	10.05%	0.17%
Xalan	7671/46	767/24	25/12	.0479	605k/267k	464k/376k	1167/450	.6064	2.40%	0.07%

That said, our use of a split-TLB is primarily an implementation decision that demonstrates how HideM can enforce fine-grained code reading policies to prevent memory disclosures. Future work will consider how to realize HideM without a split-TLB. For example, virtualization is a promising approach to enforce read policy [29]. Furthermore, HideM is similar to PAX in that we seek an implementation without CPU changes. We believe that HideM's fine-grained code reading policies are a powerful primitive that warrants investigation for inclusion into future CPU hardware designs (i.e., similar to the NX bit).

Finally, HideM assumes that adversaries have no prior knowledge of code protected in memory. This can be achieved through fine-grained binary randomization techniques better known as code diversity [15]. Fine-grained randomization can be achieved in numerous ways through both binary rewriting [34] or compiler based approaches [11]. The most efficient approaches show minimal overhead as small as 1-2% [15].

8. RELATED WORK

PaX [28] introduced non-executable page support in software using split-TLB architecture to enforce no-execute permissions on memory without support hardware. Modern hardware can enforce non-executable permissions on pages.

Van Oorschot et al. introduced bypassing self-hashing software checks via TLB de-synchronization [30]. Checksum software would read valid pages instead of modified executable pages thereby passing checks for modification. Similarly, Shadow Walker introduced stealthy hiding of rootkits through TLB de-synchronization [25]. These techniques prevent analysis of specific executable data but cannot protect the mechanism performing the de-synchronization.

Riley et al. used the split-TLB architecture to prevent code injection attacks [18]. The authors used the split-TLB to only allow memory writes to data pages. Injected code would never fill a page added to the ITLB.

HideM has similarities to a Harvard architecture where code and data are stored separately [3]. However, contemporary commodity hardware platforms implement a von Neumann memory architecture [32] where code and data are accessible in the same address-space. HideM accesses code and data in the same address-space only different data is provided for different operations.

Multics introduced execute-only memory permissions [7]. Execute-only memory could not be read by users programs. Modern binaries contain data on executable memory that make execute-only memory difficult to adopt in existing system. HideM uses both execute and read permissions for a given memory address to allow for seamless execution.

XoM encrypted executable memory which was only decrypted prior to instruction loads by a special hardware

processor [16]. Unfortunately, XoM suffers from poor performance and requires significant architectural changes. Suh et al. introduced modification to the XoM architecture to address performance issues but requires hardware architectural changes [26]. HideM is targeted at commodity systems to protect vulnerable applications from leaking data.

Research has also focused on protecting the contents of application data. Overshadow protected application data from a malicious kernel using virtualization techniques to encrypt userspace pages during kernel execution [5]. CleanOS used taint-tracking of sensitive application data in mobile devices to encrypt data that is not active [27]. VirtualGhost uses SFI [33] and CFI [2] techniques to prevent a kernel from reading application data [8]. All of these approaches do not address applications which leak their own data.

9. CONCLUSION

In this paper we presented HideM, a practical system to protect memory from leakage vulnerabilities on contemporary commodity hardware. HideM uses the split-TLB architecture to enforce fine-grained execute and read permission on memory by applying a code reading policy on memory reads and execution. The policy is generated from binary structure. HideM protects COTS and legacy binaries from disclosing critical information about memory contents. Our evaluation showed that HideM incurs limited overhead averaging 1.49% increase in runtime and 4.47% increase in memory. Furthermore, HideM significantly reduces to probability of exploitation. HideM is a practical system for enhancing the security of non-trivial applications with limited impact to performance.

10. REFERENCES

[1] 0vercl0k. Rp++ tool. `https://github.com/0vercl0k/rp`.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[3] Howard Aiken, AG Oettinger, and TC Bartee. Proposed automatic calculating machine. *Spectrum, IEEE*, 1(8):62–69, 1964.

[4] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[5] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 2–13. ACM, 2008.

[6] Tool Interface Standards Committee et al. Executable and linkable format (ELF). *Specification, Unix System Laboratories*, 2001.

[7] Fernando J Corbató and Victor A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196. ACM, 1965.

[8] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting applications from hostile operating systems. In *Proceedings of the nineteenth international conference on Architectural support for programming languages and operating systems*, 2014.

[9] Jonathon T. Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[10] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, San Jose, CA, USA, May 2014. IEEE.

[11] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[12] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *WOOT*, pages 64–76, 2012.

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

[14] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[15] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 276–291, Washington, DC, USA, 2014. IEEE Computer Society.

[16] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[17] MWR labs pwn2own 2013 write-up. MWR Labs. `https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/`.

[18] Ryan Riley, Xuxian Jiang, and Dongyan Xu. An architectural approach to preventing code injection attacks. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):351–365, 2010.

[19] Jonathan Salwan. Ropgadget tool. `http://shell-storm.org/project/ROPgadget/`.

[20] Prateek Saxena, R. Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 74–83. ACM, 2008.

[21] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[22] David Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[23] Hovav Shacham. 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*, pages 552–561. ACM, 2007.

[24] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[25] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, pages 504–533, 2005.

[26] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 339. IEEE Computer Society, 2003.

[27] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, volume 12, 2012.

[28] PaX Team. Non executable data pages, 2004.

[29] Jacob Torrey. More: measurement of running executables. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 117–120. ACM, 2014.

[30] Paul C. Van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):82–92, 2005.

[31] C+ Visual and Business Unit. Microsoft portable executable and common object file format specification, 2013.

[32] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[33] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[34] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[35] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.

[36] David L Weaver and Tom Gremond. *The SPARC architecture manual*. PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[37] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.