Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads

Adrian Tang Simha Sethumadhavan Salvatore Stolfo
Department of Computer Science
Columbia University
New York, NY, USA
{atang, simha, sal}@cs.columbia.edu

ABSTRACT

Vulnerabilities that disclose executable memory pages enable a new class of powerful code reuse attacks that build the attack payload at runtime. In this work, we present *Heisenbyte*, a system to protect against memory disclosure attacks. Central to Heisenbyte is the concept of *destructive code reads* – code is garbled right after it is read. Garbling the code after reading it takes away from the attacker her ability to leverage memory disclosure bugs in both static code and dynamically generated just-in-time code. By leveraging existing virtualization support, Heisenbyte's novel use of destructive code reads sidesteps the problem of incomplete binary disassembly in binaries, and extends protection to close-sourced COTS binaries, which are two major limitations of prior solutions against memory disclosure vulnerabilities. Our experiments demonstrate that Heisenbyte can tolerate some degree of imperfect static analysis in disassembled binaries, while effectively thwarting dynamic code reuse exploits in both static and JIT code, at a modest 1.8% average runtime overhead due to virtualization and 16.5% average overhead due to the destructive code reads.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General - *Security and Protection*

General Terms

Security

Keywords

Memory disclosure; Binary rewriting; Destructive code reads

1. INTRODUCTION

In the last decade, with the widespread use of data execution protection, attackers have turned to reusing code snippets from existing binaries to craft attacks. To perform these code reuse attacks, the attacker has to "see" the code so that she can find the "gadgets" necessary to craft the attack payload. One effective solution, until very recently, has been fine-grained randomization. The idea is to shuffle the code to blind the attacker from seeing the code layout in memory. The assumption behind this approach is that without knowledge of the code layout, the attacker cannot craft payloads. However, as demonstrated by Snow *et al.* in 2013, it is feasible and practical to scan for ROP gadgets at runtime and construct a *dynamic* JIT attack payload [22]. The attack by Snow *et al.* undermines the use of fine-grained randomization as an mitigation against ROP attacks.

To counter this new threat, researchers have revived the idea of execute-only memory (XOM) [24]. This approach involves preventing programs from reading executable memory using general purpose memory access instructions. One challenge in realizing these systems is that legacy binaries and compilers often intersperse code and data (*e.g.* jump tables) in executable memory pages. Thus, the wholesale blinding of executable memory at page granularity is not an option. To tackle this issue, researchers have used static compilation techniques to separate code and data [5]. However, this solution does not work well in the absence of source code, for instance, when utilizing legacy binaries. In fact, separating data from code has been shown to be provably undecidable [28]. Another complication in realizing the XOM concept arises from web browsers' use of JIT code where data becomes dynamically generated code. This has been shown to be a significant attack surface for browsers [1, 29].

In this work, we propose a new concept to deal with memory disclosure attacks. Unlike XOM and XOM-inspired systems, which aim to completely prevent reads to executable memory, a task beset with many practical difficulties, we allow executable memory to be read, but make them unusable as code after being read. In essence, in our model, as soon as the code is read using a general-purpose memory dereferencing instruction, the copy of code in memory is garbled. Manipulating executable memory in this manner allows legitimate code to execute without false-positives and false-negatives, while servicing legitimate memory read operations for data embedded in the code. We term our special code read operations as *destructive code reads*.

We implement our new code read mechanism by leveraging existing virtualization hardware support on commodity processors. We term our system *Heisenbyte*[1].

[1]A tribute to renowned physicist, Werner Heisenberg, who observed that the act of observing a system inevitably changes its state in quantum mechanics.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813685.

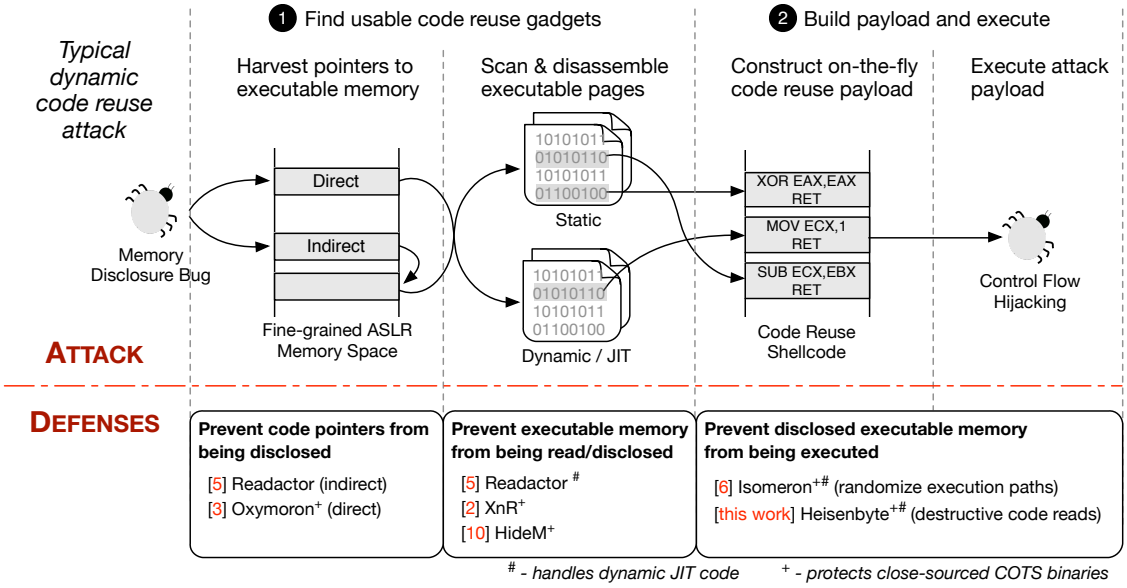


Figure 1: TOP: Stages of a code reuse attack that constructs its payload on-the-fly using executable memory found with a memory disclosure bug. BOTTOM: Taxonomy of defenses grouped by their defense strategy.

Our experiments demonstrate that Heisenbyte can thwart the use of memory disclosure attacks on executable memory, both from static program binaries and dynamically generated JIT code on a production Windows 7 machine at a modest average runtime overhead of 16.5% and 18.3% on virtualized and non-virtualized systems respectively.

Our paper makes the following contributions:

1. We conceptualize a novel destructive code read primitive that tolerates legitimate data reads in executable memory while preventing the same data from being used as code in a dynamic code reuse attack.
2. We implement Heisenbyte to realize this destructive code read operation in practice on contemporary commodity systems.
3. We demonstrate its utility in preventing attacks that use memory disclosure bugs on both static program binaries and dynamic JIT code in close-sourced COTS binaries.

The rest of the paper is organized as follows. We provide a background on the threat model in § 2. We detail the design of Heisenbyte in § 3. We describe the implementation details and challenges in § 4. We evaluate our system in § 5, and discuss the security implications and limitations of the system in § 6. We cover some related work in § 7, and conclude in § 8.

2. BACKGROUND

In this section, we describe the steps of a typical *dynamic* code reuse attack. Since the use of memory disclosure vulnerabilities is crucial in a dynamic code reuse attack (*cf.* static code reuse attacks [21, 4]), we will focus on techniques that aim to thwart executable memory disclosures. We also cover the assumptions of the threat model and the capabilities of the adversary.

2.1 Dynamic Code Reuse Attacks

In the top half of Figure 1, we show the stages of a typical code reuse attack, and the sub-steps within each stage. Typical dynamic code reuse attacks comprise two stages, namely ❶ the search for usable code reuse gadgets in either static code [22] or dynamic JIT code [1], and ❷ building the payload on-the-fly and then redirecting execution to the payload.

To gather code reuse gadgets for the dynamic exploit, an adversary needs to first uncover memory pages that are executable. Note that a trivial linear scan of the memory cannot be used as it is likely to trigger a page fault or access unmapped guard pages placed randomly in the address space. Therefore, to craft a stable exploit, the adversary has to first gather pointers to the memory pages marked as executable. These pointers can be direct branches into executable memory or indirect pointeres residing in data pages but pointing to code memory.

With the list of the pointers to executable memory, the adversary can then invoke a memory disclosure bug repeatedly (without crashing the vulnerable program) to scan and disassemble the memory pages looking for suitable code reuse gadgets. The next step involves stringing the locations of the gadgets together in an exploit payload, and finally redirecting execution to this payload using another control flow hijacking vulnerability.

2.2 Previous Works

The first category of defenses focuses on protecting the code pointers and preventing them from being disclosed, stifling the attack as earlier as possible. Oxymoron hides the direct code pointers by generating randomized code that does not have direct references to code pages [3]. However, besides using direct references to code pages, adversaries can use indirect code references that reside in stack and heap. Readactor addresses this by masking the indirect code ref-

erences with executable trampolines that are protected by hardware virtualization feature [5].

The next set of works introduces the concept of execute-only memory implemented in software. This is designed to prevent executable memory from being disclosed directly through memory read operations, consequently removing the adversary's ability to scan and locate suitable code reuse sequences for the attack. To achieve this, these works have to separate legitimate data from executable sections of programs, and distinguish at runtime between code execution and data read operations in executable memory.

XnR configures executable pages to be non-executable and augments the page fault handler to mediate illegal reads into code pages [2], but it is susceptible to disclosure attacks via indirect code references. HideM leverages the spilt-TLB architecture on AMD processors to transparently prevent code from being read by memory dereferencing operations [10]. The use of split-TLB limits its ability to remove all data from the executable sections, and inevitably exposes these data remnants to being used in attacks. Readactor relies on compiler-based techniques to separate legitimate data from code in programs and uses hardware virtualization support to enforce execute-only memory [5].

Unlike the previous defenses that *protect* the executable memory from illegal memory reads, the third group of works *tolerates* the disclosure of executable memory contents in attacks. It shifts the focus of the defense strategy to preventing any discovered gadgets from the earlier attack stages from being used in later stages of attacks. Belonging to this class of defenses, Isomeron probabilistically impedes the use of the discovered gadgets by randomizing the control flow at runtime specifically for dynamically generated code [6].

Our work also falls into this third category of defenses. While most works either enforce execute-only code memory or hide important static code contents from adversaries, we conceal the destructive changes made to executable memory (when it is read) from the adversaries. Heisenbyte allows legitimate read operations to disclose the contents of executable memory while keeping the randomized changes made to the read memory hidden.

This allows us to transparently support existing COTS binaries without the need to ensure all legitimate data and code are separated cleanly and completely in the disassembly. The heart of Heisenbyte lies on the assumption that every byte in the executable memory can only be exclusively used as code or data.

2.3 Assumptions

We assume a powerful adversary who can read (and write) arbitrary memory within the address space of the vulnerable program, and do so without crashing the program. On the target system, we also make similar assumptions used in related papers addressing the problem of memory disclosure attacks. We assume that the target system is equipped with the following protections:

- **W⊕X**: Memory pages cannot be both executable and writable at the same time. This prevents direct overwriting of existing code or injection of native code into the vulnerable program. We assume that this also applies to JIT code generated by programs, *i.e.* dynamically generated instructions cannot be executed on a memory page that is writable.

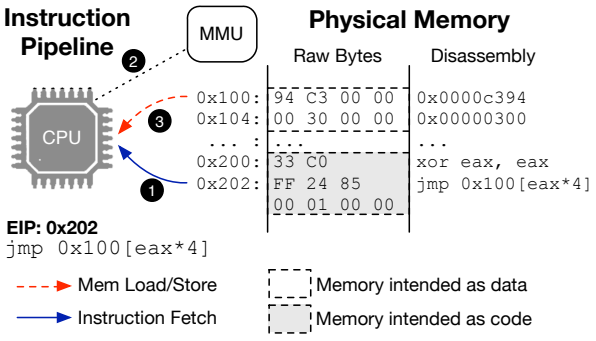


Figure 2: A typical execution of a `jmp` instruction using both code and data interleaved on the same memory page.

- **Load-time fine-grained ASLR**: All the static code from programs and libraries are loaded at random locations upon each startup. Address Space Layout Randomization (ASLR) reduces the predictability of the code layout. Furthermore, we require code layouts to be randomized at a fine granularity so that the registers [18] used and instruction locations within a function [15] or basic block [27] are different. Without this, an adversary can find code pointers in non-executable memory and infer the code layout of the rest of the memory without directly reading them.

- **Defenses against JIT attacks**: We also assume that fine-grained ASLR is applied to JIT engines [13], necessitating an adversary to perform a scan of the JIT memory pages to locate usable code reuse gadgets.

3. HEISENBYTE DESIGN

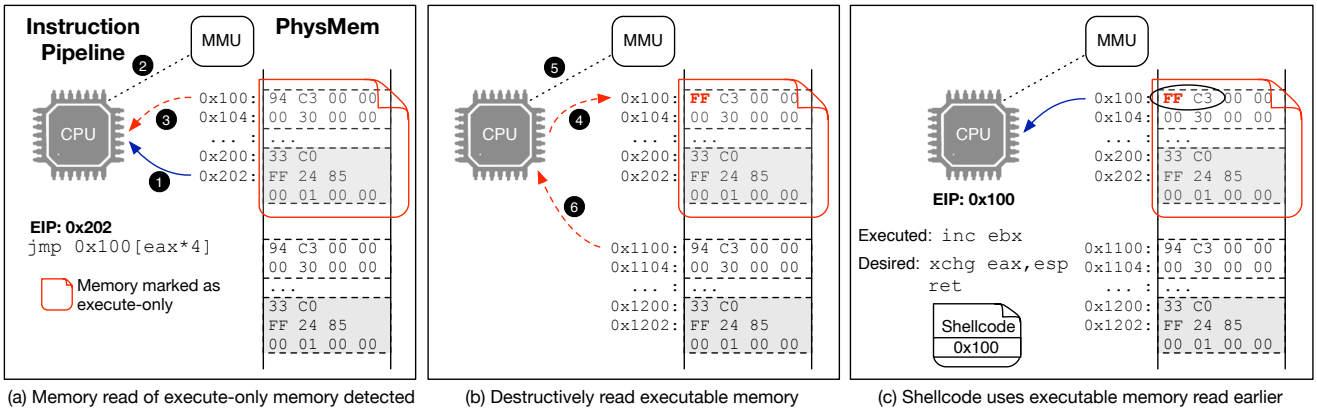
In this section, we describe our destructive code read primitive and how it thwarts memory disclosure attacks. Since our goal is to extend protection against memory disclosure attacks to COTS binaries, we also detail the challenges in determining static data from code in disassembled binaries and how they motivate our defense approach.

3.1 Destructive Code Reads

3.1.1 Review of Instruction Pipeline

We briefly review what happens in the CPU pipeline when an instruction dereferences memory for its data. This is to familiarize the reader with the distinction between a memory read or write operation that uses memory as *data* and an instruction fetch operation (which is also a special form of memory read operation) that uses memory as *code*.

Figure 2 shows the execution of a `jmp` instruction, a typical implementation of a switch statement and a very common example of both code and data residing within the same memory page marked as executable. To aid explanation, we present the raw byte representation as well as its disassembled instructions. Without loss of generality, we assume the use of 4kB memory pages for the rest of our paper. While we have demarcated the bytes that are intended to be read as data from those intended to be executed as code, note that



(a) Memory read of execute-only memory detected (b) Destructively read executable memory (c) Shellcode uses executable memory read earlier

Figure 3: Destructive code read process.

the processor is oblivious to this; all the processor knows of is the access permissions of a given memory page.

In Step ❶, the CPU performs a code fetch of the `jmp` instruction from the `0x202` address pointed to by the Extended Instruction Pointer (EIP). The instruction is decoded and the CPU determines that it needs to dereference the memory at a base address of `0x100` and an offset given by the register `eax` for its branching destination. Since the address `0x100` is in the virtual addressing mode, the CPU has to translate the address to the corresponding physical address via the Memory Management Unit (MMU) in Step ❷. For simplicity, we assume an identity mapping of the virtual to physical addresses. Subsequently, the CPU dereferences the address `0x100` via a memory load operation in Step ❸, and completes the execution of the `jmp` instruction.

3.1.2 *Destructive Code Read Process*

In Figure 3, we detail the process of how destructive code read can thwart executable memory disclosure attacks. Every Windows program binary comes with a PE header that allows us to parse and identify all static memory sections that are marked as executable. We maintain a duplicate copy of these executable memory pages to be used as data in the event of a memory read dereferencing operation. Further, in order to detect read operations in the executable memory page, we need to mark that page as execute-only.

In Figure 3(a), we show this duplicate page directly below the executable page. Like in the earlier example, the instruction is fetched at Step ❶, and the memory address of the data to be dereferenced is translated via the MMU at Step ❷. When a memory dereferencing for the data address occurs at Step ❸, this invokes a memory access violation.

The destructive code read begins at this point, shown in Figure 3(b). When we detect the read operation of the executable page, we overwrite the byte at the faulting memory address with a random byte at Step ❹. At Step ❺, via the MMU, we redirect the virtual address of the memory read to a different physical address that points to our duplicate page. We can then service the read operation transparently with the original data value at Step ❺, and the instruction that uses that data can function normally. Next, we show how these operations, specifically Step ❹, have set up a system state that can thwart a memory disclosure attack.

3.1.3 *Thwarting Memory Scan Attacks*

Since code and data are serviced by separate memory pages depending on the operation, the bytes that are read from executable memory pages may no longer be the same as the ones that can be executed at the same virtual address.

Given that a legitimate application has previously dereferenced the memory address `0x100` as data, the code memory address at `0x100` now contains a randomized byte. Executing the instruction at this address will lead to unintended operations. For instance, in Figure 3(c) if the adversary uses a memory disclosure bug to read the memory contents of `0x100`, she sees the *original* byte sequence "`94 C3`", which represents a commonly found stack pivot gadget[2]. Thinking that she has found the stack pivot gadget, she sets up her dynamic code reuse payload to use the address `0x100`. Since the earlier code read operation has "destroyed" the byte there with the random byte `FF`, when the code reuse payload executes the instruction at address `0x100`, the *garbled* byte sequence "`FF C3`" is executed as `inc ebx`. This effectively stems the further progress of the exploit.

3.2 Statically Separating Code and Data

Our use of destructive code reads in Heisenbyte at runtime is motivated by the (im)possibility of precisely and completely distinguishing disassembled bytes intended to be data from those intended to be instructions during runtime. This leads us to adopt a fundamentally different strategy from the earlier works that enforce execute-only memory using compiler-based techniques. Instead of determining the code or data nature of bytes during offline static analysis and enforcing runtime execute or read policies based on this, we infer the code/data nature of bytes at runtime, identify the inferred data bytes in executable memory, and remove the possibility of using them as executable code in attacks. We describe some of the main challenges of accurately identifying data in executable sections of Windows binaries, and how we sidestep these challenges using binary rewriting.

3.2.1 *Challenges in Distinguishing Data from Code*

Halting Problem Legitimate data must be separated out from the disassembled bytes of the executable sections

[2]A sequence of instructions modifying the stack pointer to address a code location of the adversary's choosing

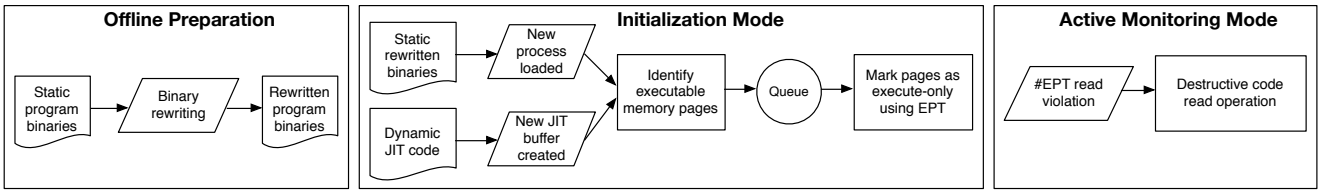


Figure 4: Flowchart of configuration of EPT for monitored executable pages.

of the binaries. To do so requires making a judgment on whether or not a range of bytes is intended to be used as data at runtime. While heuristics can be used to make that judgment, this code or data separation task at binary level essentially reduces to the halting problem because we can be sure only at runtime when bytes are truly intended to be code, and yet we want to do this during static analysis [28].

JIT Code Generation Web scripting languages such as Javascript are optimized for efficient execution by modern web browsers using just-in-time compilation. While the newer versions of web browsers like Internet Explorer and Mozilla Firefox separate the code and data into different memory pages, with the latter in non-executable ones [1], the older versions however emit both code and data on same executable pages. We want to support the use of these legacy JIT engines.

Corner Cases In our analysis of Windows shared libraries, we found that there are many corner cases where the disassembler cannot accurately determine statically if a chunk of bytes is intended to be data or code. This stems from the limitations of the disassembly heuristics used by the disassembling engine.

A common example of incorrect disassembly is the misclassification of isolated data bytes as `RET` return instructions within a data block. A `RET` instruction is represented in assembly as a one-byte opcode, and can potentially be a target of computed branch instructions whose destination cannot be statically decidable. Therefore, the disassembler frequently misclassifies data bytes that match the opcode representation of return instructions as code.

We also found situations which assume that code and data sections are located in a specific layout. For example, in `kernel32.dll`, a shared library used by all Windows binaries, the relocation section indicates a chunk of bytes that are dereferenced as data at the base of the executable `.text` section. Because a readable and writable data section `.data` almost always follows this `.text` section, any instruction referencing this data also assumes that 400 bytes following this address has to be a writable location. This structural assumption is extremely difficult to discern during offline static analysis. If we blindly relocate this data from the executable `.text` section to another section without respecting this structural assumption, a crash is inevitable.

3.2.2 *Our Conservative Separation Approach*

As mentioned previously legacy COTS binaries, especially Windows native programs and libraries, have substantial amount of legitimate data interleaved with code in the executable sections. Blindly retaining these data can lead to exhorbitant overheads in Heisenbyte as read access to each of the these data items in the executable memory will incur the overhead of the destructive code read operation.

To mitigate these overheads, we perform very conservative static analysis to determine well-defined data structures that can be safely relocated out of the executable sections without affecting the functionality of the program. For instance, in many legacy Windows binaries, the read-only data sections are merged with the code section. This is not a problem because the format for the data section is well-documented. Similarly, we also handle well-structured data chunks like strings, jump tables and exception handling information. Here, we describe examples of these legitimate data chunks commonly interspersed with code in the executable sections of Windows COTS binaries.

Standard data sections Many Windows native binaries have the standard non-executable data-only sections embedded within the executable `.text` section. Examples include the Import Address Table, the Export Address Table and debug configuration section.

Merged data sections An optimization technique to minimize the file sizes of programs is to merge the read-only data section (`.rdata`) and the main executable section (`.text`)[3]. This technique is commonly used in Windows native binaries and shared DLL libraries. We are specifically targeting the relocation of two types of read-only data in this section, namely strings and Structured Exception handler (SEH) structures, since they are well defined.

Jump tables High-level switch statements are implemented as jump instructions and jump tables in assembly. Compilers typically position the jump table offsets near the jump instructions that use jump tables. These jump tables are intended to be dereferenced as data at runtime.

4. SYSTEM IMPLEMENTATION

In this section, we detail the various components of Heisenbyte, and how we realize the mechanism of destructive code reads on selected executable memory pages. As shown in Figure 4, we achieve this in three different stages. We begin by rewriting the program binaries that we want to protect to separate specific data from the code in an **Offline Preparation** stage. We detail this process in § 4.1.

To ensure that our destructive read operations only apply to the processes we want to protect, Heisenbyte processes targeted executable memory pages in the following two modes. We discuss each of them in detail in § 4.2.

- **Initialization mode**: This mode identifies at runtime selected executable memory pages to protect, and subsequently configures execute-only access permissions for these pages, in preparation for the next mode.

[3]This can be achieved using Microsoft Visual Studio compiler with the linker flag `/merge:.rdata=.text`.

- **Active monitoring mode**: Once the set of executable pages is configured with the desired EPT permissions, this mode is then responsible for performing the destructive code read operation when it detects a read operation to an executable page.

Furthermore, to demonstrate that the technique is practical on COTS binaries, we invest substantial effort in this work to develop Heisenbyte to work on the primarily close-sourced Windows OS. The techniques and design presented in this work can be generalized to other OSes like Linux.

4.1 Offline Static Binary Rewriting

Recognizing well-defined data in disassembly We use the state-of-the-art commercial disassembler, IDA Pro, to generate the disassembled code listing of the programs. We also leverage IDA Pro's built-in functionality to identify well-defined data structures (described in earlier sections) commonly found in executable memory pages.

Rewriting engine We develop our binary rewriting engine as a Python script. Unlike traditional binary rewriting tools, we do not perform any rewriting operations that change the semantics of instructions. Our engine focuses on using disassembly information from IDA Pro and the section headers to determine if a range of bytes within an executable section needs to be relocated to a separate data section. Our engine reconstructs the PE header to add a new non-executable section to consolidate all these identified data. Relocation information is crucial in aiding both our static analysis and our relocation operations. For example, if a range of data bytes needs to be relocated to another section, the relocation table is updated either by adding new relocation entries or editing existing ones to reflect the new location of the relocated data. Relying on the relocation tables allows us to transparently move bytes around within a PE file without breaking the functionality of the program.

Overcoming Windows binary protection To evaluate our rewritten Windows native library files with Heisenbyte, we need to replace the original files. However, on Windows, critical shared libraries and program binaries are protected by a mechanism called Windows Resource Protection (WRP) [17]. WRP prevents unauthorized modification of essential library files, folders and registry entries by configuring the Access Control Lists (ACLs) for these protected resources. Only the Windows Installer service, *TrustedInstaller*, has full permissions to these resources.

To get around this problem, we rely on the fact that we have administrative privileges on the system. We take control of the ownership of the protected files from the *TrustedInstaller* account using the command `takeown.exe`, and grant to our account full access rights for the protected files using `icacls.exe`. At this point, we can rename the files but we cannot replace the files because they are still in use. We rename the files and copy our rewritten binaries with the original filename. When the system is rebooted, our rewritten libraries will be then loaded into the system. To ensure integrity of the binaries, the modified ACLs of the protected binaries are restored after the rewritten binaries are replaced.

This technique of deploying rewritten Windows native files work for most of the binaries with one exception – `ntdll.dll`. The integrity of this file is verified when the system starts up. We solve this by disabling the boot-time integrity in the bootloader [11], so that the rewritten `ntdll.dll` binary can be loaded.

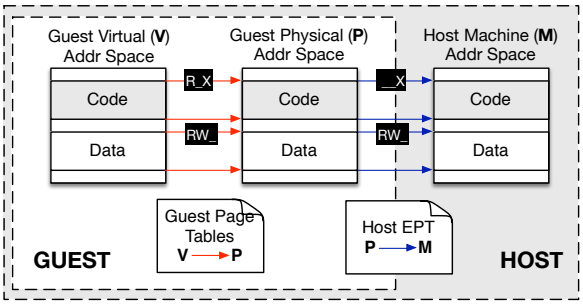


Figure 5: Nested paging structure using virtualization hardware support (using Intel-specific terms).

4.2 Heisenbyte Core Monitoring Components

4.2.1 *Review of Intel Extended Page Tables (EPT)*

Before we discuss each of the components in the two modes, we first describe the key hardware virtualization feature we use to achieve our goals.

Heisenbyte needs to be able to detect when executable memory is being read. There are a number of ways to do this: mediating at the page fault handler [2] or leveraging the split-TLB microarchitecture of systems [10]. These solutions stem from the limitation of current OSes not being able to enforce execute-only permissions on memory pages. Fortunately, hardware virtualization support – hardware-assisted nested paging – on commodity processors provides a means for us to enforce fine-grained execute-only permissions on memory pages. This hardware feature augments existing page walking hardware with the ability to traverse in hardware the paging structures mapping guest physical (`P`) to host machine (`M`) addresses. This eliminates the overhead involved in maintaining shadow page tables using software. A virtualization-enabled MMU maps virtual (`V`) addresses in the guest to machine physical addresses in the host, using both the guest page tables and the host second-level page tables[4]. This is done transparently of the guest OS.

We show three address spaces spanning across the guest and host modes in Figure 5. In the guest, the page tables store the `V`→`P` address mappings, as well as the corresponding permission bits. These guest page tables, described earlier, cannot be configured with solely the execute bit set. Conversely, in the host, the EPTs maintain the `P`→`M` address mappings. The key difference between the EPTs and guest page tables is that the EPTs can configure each page mapping as execute-only. When an access to a memory page violates the permissions configured for that page, an `#EPT` violation is invoked, transferring control to the hypervisor.

This mechanism is instrumental in our system to detect read operations to executable memory. In our work, like Readactor [5], we rely on hardware-assisted EPT to configure guest physical memory pages as execute-only with no read or write access. Since this is a virtualization-assisted technology, virtualization has to be enabled on the system

[4]Intel terms this Extended Page Tables(EPT), and AMD calls this Nested Page Tables (NPT)

we are trying to protect. On systems that need to protect existing virtualized guests, Heisenbyte can be implemented within the Virtual Machine Monitor (VMM) software, such as Xen or KVM. However, the need for virtualization does not preclude the protection of non-virtualized systems.

To demonstrate this, we make a conscientious effort to implement Heisenbyte for a non-virtualized OS. We develop Heisenbyte as a Windows driver that will configure the EPT paging structures, enable virtualization mode and place the execution of the non-virtualized OS into virtualized guest mode (non-root VMX mode). Heisenbyte does this on a live running system, without requiring any system reboot. The *host mode component* (shown in Figure 6) of our driver ensures that the running system functions as usual, by configuring the EPT structures to use identity mappings from the guest physical to host machine addresses. At this point, our host mode component is in a position to configure the execute-only permissions transparently of the guest OS.

4.2.2 Identifying Executable Memory

Before we can configure the EPT execute-only permissions, we need to first identify which executable memory pages to monitor. To do that, we have to track when and where executable memory from processes are loaded and mapped. Since the treatment of dynamic code tracking is more involved, we will describe them in detail separately.

Static program binaries To deal with static code, Heisenbyte *guest mode component* (as shown in Figure 6) begins its initialization by registering Windows kernel-provided callback functions associated with the creation/exiting of processes and loading/unloading of shared libraries. Using the callback registration APIs, `PsSetCreateProcessNotifyRoutine` and `PsSetLoadImageNotify`, our driver guest component is informed whenever a new static code process or library gets loaded. This callback mechanism applies to both executable files and shared library files. If a newly loaded static image matches within a whitelist of binaries we are protecting, our guest mode component parses the memory-mapped PE header to get the list of guest virtual addresses and sizes of the executable sections in each loaded image.

With the guest virtual addresses, we need to retrieve the corresponding guest page table and guest physical addresses for each virtual memory page to configure the EPT entries. However, since the OS performs a lazy allocation when doing the memory mapping, these memory pages may not be paged into memory yet. As a workaround, Heisenbyte schedules a thread within the context of the target process and accesses one byte in each memory page to invoke the paging-in mechanism. Further, Heisenbyte uses the `MmProbeAndLockPages` kernel API to make the pages resident in the physical memory, so that they cannot be paged out. This necessarily increases the memory working set of a program. We will investigate this in § 5.2.2.

These information is stored in a queue buffer shared by the guest mode and host mode components. It is noteworthy that since the guest mode component runs in the VMX non-root guest mode, it has no access to the EPTs. The configuration of the EPT mappings has to be performed by the host mode component.

Dynamic JIT code Unlike the loading of static binaries into memory, dynamic memory buffer creation/freeing does not have convenient kernel-provided callbacks. Furthermore,

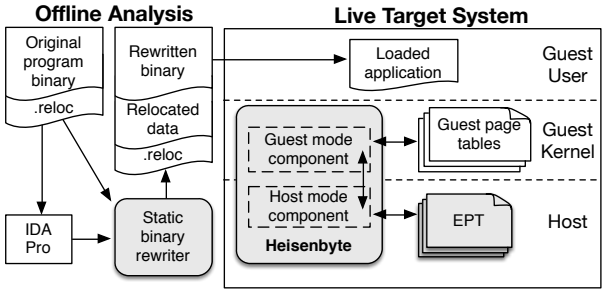


Figure 6: Overview of system architecture (Heisenbyte components are shaded grey).

the protection bits of a dynamic buffer may change at runtime during the generation and execution of dynamic code. For example, a modern JIT-enabled browser, like Safari, first allocates a writable (read/write RW) buffer as a code cache to fill with generated native code. With our assumption that hardware W⊕X DEP is enforced, the JIT engine has to remove the writable permission and make the code cache executable (read/execute RX) before executing the code cache. If the dynamic code cache subsequently needs to be modified, the buffer is restored to a writable (read/write RW) one before changes to the code cache can be made.

Based on the lifetime of the buffer during which the code is ready to be executed, we observe that we only need to monitor the buffer during this period of time. Specifically, we begin tracking a dynamic buffer when the protection bits changes from non-executable to executable, and stop tracking a dynamic executable buffer when it is freed or when its executable bit is removed.

Windows-specific implementation Next we discuss how we detect when dynamic memory buffers are turned executable and when they are freed. All operations that are used to free or change protection bits of memory result in two functions in `ntdll.dll`, `NtFreeVirtualMemory`, and `NtProtectVirtualMemory` respectively, just before invoking the system calls to the kernel services. When `ntdll.dll` is loaded into our target process, we modify the entry points of these two functions with trampolines to a Virtual Memory (VM)-tracking code that resides on a dynamically allocated page. Since the function hooking is performed in-memory, the OS Copy-on-Write mechanism ensures that these hooks only apply to the target process.

In practice, dynamic memory buffers are created and freed very frequently. Since we are only interested in executable buffers, we use an auxiliary bitmap data page to indicate if an executable buffer of a given virtual address has been previously tracked. This added optimization enables the VM-tracking code to decide if it should handle specific events.

The VM-tracking code that monitors the changing of protection bits of buffers performs a hypercall to our host mode component whenever an executable buffer is configured to be non-executable and vice versa. The host mode component updates the address bitmap depending on whether a new executable page is being tracked or removed from tracking. Conversely, the VM-tracking code that monitors the freeing of executable buffers will perform a hypercall when it determines from the bitmap that a buffer with a given virtual address is being freed. The host mode component will then

reset the EPT mapping for the physical pages of the buffer to an identity mapping, essentially stopping the tracking of this dynamic executable buffer.

Protecting VM-tracking code and data The VM-tracking code resides on a dynamically allocated executable page, and is protected by Heisenbyte just like any typical executable memory page. Conversely, by being configured to be read-only from the userspace, the auxiliary bitmap is protected from any tampering attacks originating from the userspace; it can only be modified in the host kernel mode (specifically by the host mode driver component. Furthermore, a XOR-based checksum of the bitmap is maintained and verified before the bitmap is updated in the host mode component.

4.2.3 *Overcoming Challenges in using EPT*

Problem of shared physical memory pages One key challenge in using EPT to enforce execute-only memory is that the guest physical memory pages may be shared by multiple processes due to the OS's Copy-on-Write (COW) optimization. This COW mechanism is a common OS optimization applied to static binaries to conserve physical memory and make the startup of programs faster. Thus the OS lazily duplicates the original page into a newly allocated physical page only when the process writes to the memory page. Before these physical memory pages are duplicated by COW, they are shared by multiple processes. Enforcing execute-only permissions on these shared guest physical pages may result in many `#EPT` violations triggered by processes we do not care about and cause unnecessary overhead.

Inducing COW on physical pages Heisenbyte overcomes this problem by inducing COW on the executable memory pages of target processes. We leverage the guest OSes' innate COW capability to transparently allocate new physical memory pages for the static code regions of processes we want to protect. To invoke COW on the memory pages of processes, the write operation must occur in the context of the process; a write operation originating from the hypervisor into the memory space of a user process will not trigger the copy-on-write mechanism.

When a static binary is loaded into memory, Heisenbyte schedules an Asynchronous Procedure Call thread [16] to execute in the context of the target process. This thread suspends the execution of the original target process, enumerates the static code regions of the process using the PE headers mapped in the address space, and performs a read and write operation on each executable memory page. This identity-write operation is very efficient since we only "touch" one byte in each 4kB memory page. The OS detects this memory write and invokes the COW mechanism. In this manner, each executable static page in a process will no longer share a physical page with another process.

The executable memory pages are then configured to be read-only using EPT by the host mode component only after the COW-inducing thread has completed processing all the executable memory pages of the newly loaded binary.

4.2.4 *Intervention with Code Garbling*

Maintaining separate code views To enable our destructive code read operations while allowing legitimate data reads in executable memory to function properly, we need to maintain separate code and data views for each executable memory page we are protecting. We leverage the EPT to transparently redirect the use of any guest virtual address to the desired view at runtime. In Figure 7(a), before a target process is being protected, an identity EPT mapping of the guest physical to host machine memory is maintained.

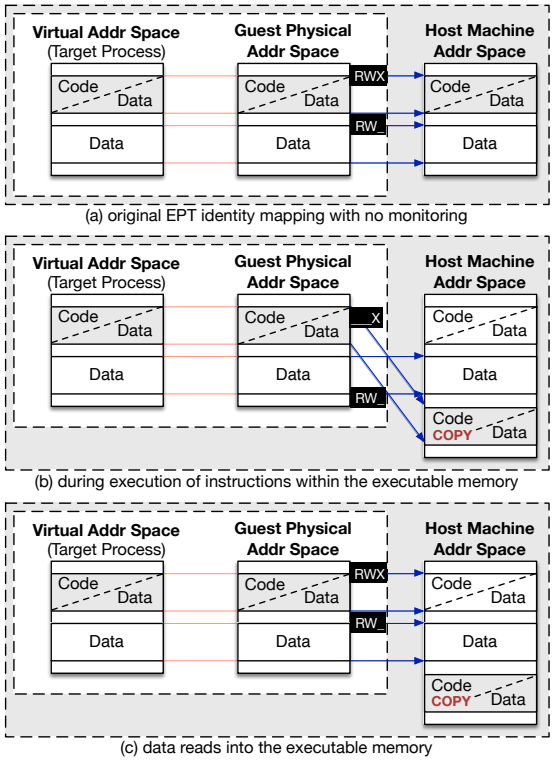


Figure 7: Using EPT to maintain separate code and data views transparently.

After identifying the guest physical memory pages to protect, we add a duplicate page in the host machine address space. Any subsequent instructions being executed are redirected to the *code copy* memory page shown at the bottom of Figure 7(b). This guest physical page is configured to be execute-only using EPT.

Destructive reads into executable memory With the executable pages configured to trigger a VM exit upon a data read, our `#EPT` violation handler in the host mode component of the driver can intervene and mediate at these events. At each `#EPT` read violation, we overwrite the data read address within our *code copy* page with a random byte. This constitutes the destructive nature of our code reads. Since there are legitimate data reads into executable memory from the kernel, especially during PE loading, we perform the byte garbling only when the read operation originates from user-space.

Next we edit the EPT entry to have read/write/execute access and redirect the read operation to read from the original code page, now intended exclusively to service data read requests, as shown in Figure 7(c). To restore the memory protection, we set the single-step trap flag in the EFLAGS so that a VM exit is triggered immediately after the instruction performing the read operation. At this point, we restore the EPT permissions to execute-only to resume operation.

5. EVALUATION

In this section, we demonstrate the utility of Heisenbyte in stopping attacks that use static and dynamic memory disclosure bugs. We evaluate the performance and memory overhead of our system. Our experiments are done on 32-bit Windows 7 running on a quad-core Intel i7 processor with 2GB RAM. As our prototype does not handle SMP systems, we configure the system to use only one physical core.

5.1 Security Effectiveness

5.1.1 Memory Disclosure Attack on Static Code

We use the Internet Explorer (IE) 9 memory disclosure vulnerability (CVE-2013-2551) presented by Snow *et al.* [22]. This is a fairly powerful heap overwrite vulnerability involving a Javascript string object. It enables an adversary to perform arbitrary memory read and write operations repeatedly without causing IE to crash. On our test setup, we craft an exploit that leverages this memory disclosure bug as a memory read and write primitive.

As ASLR is enabled by default – Window's ASLR is a coarse-grained form that changes only the base addresses of the shared libraries at load time –, the exploit has to look for suitable code reuse "gadgets" to string together as an attack payload. To demonstrate that our system works with an exploit that uses disclosed executable memory contents, we craft our exploit to dynamically locate a stack pivot ROP gadget.

The exploit begins by first leaking the virtual table pointer associated with the vulnerable heap object. This pointer contains an address in the code page of `VGX.dll` shared library. Using the memory read primitive, the exploit scans backwards in memory for the PE magic signature `MZ` to search for the PE header of the shared library.

It is noteworthy that at this point, if IE uses any code within the range of bytes the exploit has scanned, IE will crash due to the corruption of legitimate code by the destructive code reads. However, in a real deployment, as defenders, we do not want to rely on such opportunistic crashes. We assume that the exploit avoids scanning executable memory during this stage and only reads non-executable memory.

When the exploit finds the PE header of the library, it can then derive the base address of `user32.dll` by parsing the import address table in the PE header. The shared library `user32.dll` contains a set of ROP gadgets that are found offline. With this, the exploit can construct its ROP payload by adjusting the return addresses of the pre-determined ROP gadgets with the base address of `user32.dll`. To simulate the dynamic discovery of "gadgets" in a dynamic code reuse exploit, we craft the exploit to perform a 4-byte memory scan at the location of the stack pivot gadget, and then redirect execution to that stack pivot gadget.

While our actual system uses a randomized byte to garble the code, we use a fixed `0xCC` byte (*i.e.* a debug trap) for the code corruption in this experiment. This allows us to be sure that any crash is directly caused by our destructive code reads. When control flow is redirected to the stack pivot gadget, IE crashes at the address of the stack pivot with a debug trap. This demonstrates that Heisenbyte stems the further progress of the exploit as a result of corrupted byte caused by the exploit's executable memory read.

Furthermore, we configure the `Windbg` debugger to automatically launch upon application crash. When the debugger is invoked at the crash address at the location of the stack pivot, the debugger displays and disassembles the original byte sequence of the stack pivot gadget in `user32.dll`. As the debugger reads memory as data read operations, the original bytes at that code address are shown. It is apparent that what gets *executed* is different from what gets *read*. This further demonstrates that Heisenbyte correctly maintains separate code and data views of executable memory.

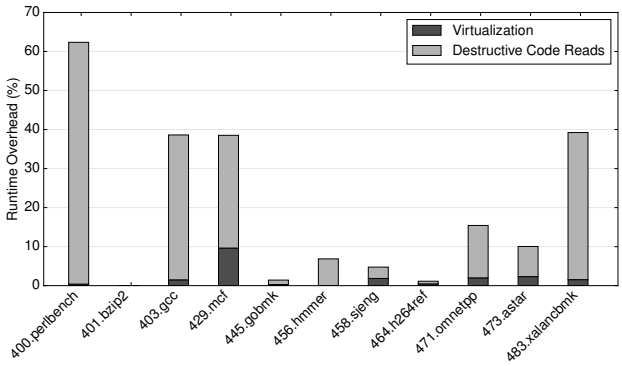


Figure 8: SPEC2006 execution overhead.

5.1.2 Memory Disclosure Attack on Dynamic Code

At the time of writing, we are aware of only one publicly available exploit [19] that uses an integer overflow bug to achieve memory read/write capability on the JIT code cache of mobile Chrome. However, this exploit only works on ARM devices, so we cannot use this for our evaluation.

To evaluate our system on memory disclosure attack on dynamically generated code, we create a vulnerable program that mimics the behavior of JIT engine in the creation of dynamic executable buffers. Our program allocates a readable and writable buffer and copies into this buffer a pre-compiled set of instructions that uses a jump table. This is similar to the behavior of legacy JIT engines that emit native code containing both code and data in the dynamic buffer.

With the code cache ready to execute, our program makes the dynamic buffer executable by changing the permission access to readable/executable, and executes the buffer from the base address of the buffer. The program functions correctly with Heisenbyte running. Since the jump tables in the dynamic buffer are only ever used as data in the lifetime of the buffer, Heisenbyte properly supports the normal functionality of the simulated JIT-ed code.

To simulate an attack that scans the memory of the dynamic code region for code reuse gadgets, we create an exploit to leverage a memory disclosure bug we have designed into the program. The exploit uses this bug to read the first four bytes of the dynamic buffer and redirects execution control to the start of the dynamic buffer. Like in the case of the experiment with IE9, the vulnerable program crashes at the base address of the dynamic buffer as a result of the destructive code reads induced by Heisenbyte.

5.2 Performance Overhead

5.2.1 Execution Overhead

We measure the slowdown caused by various components of Heisenbyte using the SPEC2006 integer benchmark pro-

grams. Since our solution works on and rewrites binaries, we first compile the programs and work with the compiled binaries assuming no source code is available. We compile the SPEC2006 programs with Microsoft Visual Studio 2010 compiler using the default linker and compilation flags. As the compiler does not support the C99 feature, *e.g.* type `_complex`, we cannot successfully compile `462.libquantum`. We thus use only 11 out of 12 SPEC2006 integer applications for our evaluation. For all the tests, we restart each set of runs on a rebooted system, perform 3 iterations using the base reference input and take the median measurements.

We evaluate the execution slowdown caused by Heisenbyte to an originally non-virtualized system. The overhead of Heisenbyte comprise two main sources, namely the overhead as a result of virtualizing the entire system at runtime, and the overhead of incurring two VM exits for each destructive code read operation. Separating the measurements for the two allows us to evaluate the overhead net of virtualization when Heisenbyte is deployed on existing virtualized systems (they are already occurring the virtualization overhead).

To measure the overhead caused by purely virtualizing the system, we run the SPEC benchmarks with the Heisenbyte driver loaded, but without protecting any binaries or shared libraries. Compared to a baseline system, the virtualization overhead ranges from 0% (`401.bzip2`) to 9.6% (`429.mcf`). The virtualization overhead is highly dependent on the execution profile of the programs. We attribute the high overhead for `401.bzip2` to the paging operations performed by Intel EPT hardware page walker. On average, the geometric mean of the virtualization overhead caused by Heisenbyte is 1.8% across all the programs.

With the measurements for the virtualization overhead, we can now measure the overhead of the destructive code reads due to the incomplete removal of legitimate data from the executable memory pages. We configure Heisenbyte to protect the SPEC binaries and all the shared DLL libraries used by SPEC, and compare the execution time to the baseline. The variance in this overhead is huge, depending on how much legitimate data is not removed by the binary rewriting. The destructive code read overhead ranges from 0% (`401.bzip2`) to 62% (`400.perlbench`), with an average of 16.5% across the programs. This overhead is a direct consequence of the imperfect removal of legitimate data from the executable memory pages at the binary rewriting stage. The higher the frequency a program accesses such legitimate data in the memory pages, the greater the overhead incurred by the destructive codes. The average of the combined virtualization and destructive code read overhead is 18.3%.

In this work, we choose to be very conservative in the types of data that we relocate out of the executable sections during the binary rewriting to show that the system can still tolerate the incomplete relocation of *all* data from the executable sections. This overhead can be further reduced with a more aggressive strategy in removing the data.

5.2.2 *Resident Memory Overhead*

As discussed in § 4.2.2, Heisenbyte requires keeping the executable memory pages resident in physical memory when configuring the EPT permissions and monitoring for data reads to these pages. Here we evaluate how much more physical memory overhead introducing Heisenbyte causes. We measure this by tracking the *peak* Resident set size (RSS) of a process over entire program execution. RSS measures

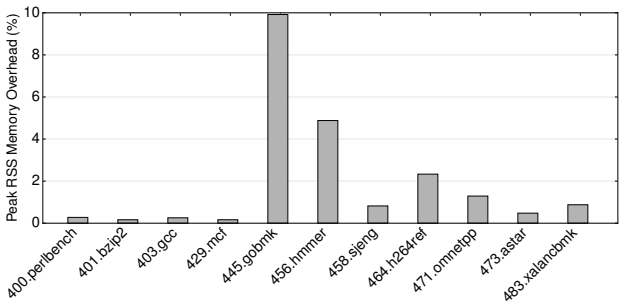


Figure 9: Memory overhead in terms of peak RSS.

the size of process memory that remains resident in the RAM or physical memory. We inject a profiling thread to our processes to log the current maximum RSS as the process runs every 20 seconds. Figure 9 shows a modest increase of 0.8% on average in the peak RSS across all the programs.

6. DISCUSSION

Code leaks via side channels While Heisenbyte thwarts the use of the disclosed gadgets found by *directly* scanning executable memory with a memory disclosure bug, it does not protect against attacks that *indirectly* leak the locations of code reuse gadgets through side channels, such as timing channels [20, 9]. Comprehensive protection against side channel leaks is generally recognized as a prohibitively challenging tasks in the general context, and not just pertaining to the disclosure of executable memory.

Our work focuses on protecting client-side COTS binaries prevalent on Windows systems. Most of these programs are not tolerant of crashes. Furthermore, exploiting these user applications is time-sensitive. For example, an attacker loses the opportunity to exploit the system once its exploit invokes a crash on IE or takes too long. These aforementioned reasons make existing side channel-based memory disclosure attacks on this class of binaries challenging. Therefore, we do not consider this type of attacks in our work.

Size of garbled code At present, Heisenbyte disregards the operand size of the instruction performing the reads into the executable memory, and performs destructive code reads of only one byte. An adversary who uses data reads of four bytes to scan the memory can potentially exploit this. Garbling only one byte will give the adversary the potential to use the remaining three bytes from the data reads. To tackle this problem, Heisenbyte can easily be extended to handle code reads using different operand sizes. We can maintain three hashtables, each storing the opcodes used for 1-byte, 2-byte and 4-byte operands. Whenever a code read happens, Heisenbyte can look up the hashtable to determine efficiently the size of operand and destroy the same number of bytes accordingly.

Support for fine-grained ASLR Heisenbyte requires fine-grained ASLR to ensure that the layout of code cannot be inferred with partial reads into the non-executable sections [20, 9]. Fine-grained ASLR can be extended in Heisenbyte in a number of ways. For example, since we are rewriting the binaries, fine-grained ASLR such as in-place code randomization [18] can be extended into the rewriting process. As no additional code is introduced, such in-place code

randomization have limited impact on code locality and, as past research has shown, incurs negligible runtime overhead.

New hardware features to reduce overhead In this work, we choose to implement Heisenbyte with the standard virtualization features found in most processors. The goal is to provide a baseline proof-of-concept implementation of our design. As we have seen in § 5.2, the major source of overhead comes from inducing the VM exits to implement the destructive code reads. This can be reduced substantially with the combined use of two new virtualization features in the recent Haswell processor [14]. This processor allows selected `#EPT` violations to be converted to a new type of exception that does not require VM exits to the hypervisor. The latency of VM exits can then be reduced substantially. This exception is known as the `#VE` Virtualization Exception. With this feature, during the active monitoring mode, a data read into protected executable memory pages will trigger an exception and control will be handed over to the guest OS `#VE` Interrupt Service Handler (ISR). To handle the configuration of EPT entries, the second feature, named EPT Pointer switching, allows the guest OS to efficiently select within a pre-configured set of EPT pointers having the required EPT permissions we need.

Code read logs to guide binary rewriting As an optimization to aid the offline static analysis, we can augment Heisenbyte to record all read operations into executable memory into a log buffer. This log can then be used to direct the static analysis in determining if a set of bytes within an executable section is indeed intended as data at runtime. The binaries can be analyzed and rewritten repeatedly using this information to achieve a high code coverage over time. This can further reduce the overhead of the system, since the data reads that previously trigger VM exits will no longer occur.

Graceful remediation In additional to detecting attacks, Heisenbyte can offer the capability to gracefully terminate, instead of crashing, the process that is being targeted by the attack, and provide further alerting information regarding the attack to the user. Instead of using randomized junk bytes for the destructive code reads, Heisenbyte can use specific bytes designated to induce selected software interrupts or traps when executed. The host component of Heisenbyte can be configured to mediate on these interrupts. When malicious code attempts to execute code modified by earlier reads, pertinent information about the attempted code execution, such as the faulting instruction, and the original and modified contents of the executable memory page, can then be logged. This may assist in identifying the associated vulnerability, and provide useful forensics information for vendors to patch the program.

7. RELATED WORK

Our work is enabled by two key techniques, namely the ability to maintain separate code and data views in a von Neumann memory architecture[5], and destructive read operations applied on executable memory. We have described the research works most closely related to our work in § 2.2. Here we detail other works using the above two techniques.

Maintaining separate code/data views Many have explored the value of maintaining separate views for code and data. The earliest works are mostly offensive in nature. Van Oorschot *et al.* leverage the process of desynchronization the TLB to bypass self-hashing software checks [26]. Shadow Walker, a rootkit, relies on the split-TLB architecture of processors to hide its malicious code from being detected by code scans by Antivirus [23]. Torrey explores the use of EPT to differentiate code from data at runtime to perform attestation on dynamically changing applications [25]. Spider also uses EPT permissions to maintain different views for code and data to implement the evasion-resistant breakpoints that are "invisible" to the guest [7]. Our work shares similar EPT-based techniques with some of these works, albeit towards vastly different objectives.

Destructive reads Examples of destructive read operations in practice are sparse. The destructive-read embedded DRAM [8] is a special-purpose DRAM that allows destructive reads to conserve power consumption. The contents of the memory can only be read once. At the software level, destructive read operations are sometimes performed by the BIOS during the memory check in its Power-On Self Test (POST), with the purpose of ensuring sensitive memory contents cannot be leaked [12]. Our software-emulated destructive read primitive on executable memory represents the first work to apply this technique to make system states non-deterministic and harder for adversaries to exploit.

[5]where code and data are stored in the same addressable memory

8. CONCLUSIONS

We present the novel use of destructive code reads to restrict adversaries' ability to leverage executable memory that are exposed using memory disclosure bugs as part of an attack. We realize this technique in Heisenbyte using existing hardware virtualization support to identify read operations on executable memory. To date, Heisenbyte is the first system that guarantees the disclosed executable memory cannot be executed as intended, while still tolerating some degree of data not removed from the code pages. Our experiments demonstrate that Heisenbyte prevents the use of disclosed executable memory in real and synthetic attacks, while offering transparent protection for legacy close-sourced binaries, at modest overall runtime overheads averaging 18.3%. Amongst defenses that work on breaking determinism in systems, Heisenbyte represents a resolute and effective step towards stopping advanced exploits.

Acknowledgments. We thank the anonymous reviewers for their feedback on this work. This work is supported by grants FA 865011C7190, FA 87501020253, CCF/SaTC 1054844 and a fellowship from the Alfred P. Sloan Foundation. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or commercial entities.

9. REFERENCES

[1] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*, 2015.

[2] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. 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*, CCS '14, pages 1342–1353, New York, NY, USA, 2014. ACM.

[3] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. *Proc. 23rd Usenix Security Sym*, pages 433–447, 2014.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[5] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[6] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. 2015.

[7] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 289–298, New York, NY, USA, 2013. ACM.

[8] H. Dybdahl, P. G. Kjeldsberg, M. Grannæs, and L. Natvig. Destructive-read in embedded dram, impact on power consumption. *J. Embedded Comput.*, 2(2):249–260, Apr. 2006.

[9] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[10] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 325–336, New York, NY, USA, 2015. ACM.

[11] Fyyre. Disable patchguard - the easy/lazy way. http://fyyre.ivory-tower.de/projects/bootloader.txt, 2011.

[12] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[13] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 993–1004. ACM, 2013.

[14] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3C, 2014.

[15] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. 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.

[16] Microsoft. Asynchronous procedure calls. https://msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx.

[17] Microsoft. Windows resource protection. https://msdn.microsoft.com/en-us/library/windows/desktop/cc185681(v=vs.85).aspx.

[18] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615. IEEE, 2012.

[19] P. Pie. Mobile Pwn2Own Autumn 2013 - Chrome on Android - Exploit Writeup, 2013.

[20] J. Seibert, H. Okkhravi, and E. Söderström. 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*, pages 54–65. ACM, 2014.

[21] H. 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.

[22] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[23] S. Sparks and J. Butler. Raising the bar for windows rootkit detection. http://phrack.org/issues/63/8.html, 2005.

[24] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 168–177, New York, NY, USA, 2000. ACM.

[25] J. Torrey. More shadow walker: Tlb-splitting on modern x86. Blackhat USA, 2014.

[26] P. Van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):82–92, April 2005.

[27] R. Wartell, V. Mohan, K. W. Hamlen, and Z. 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.

[28] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham. Differentiating code from data in x86 binaries. In *Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.

[29] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*, 2015.