Computers & Security 132 (2023) 103377

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

What you can read is what you can't execute

YongGang Li[a,*], JiaZhen Cai[a], Yu Bao[a], Yeh-Ching Chung[b]

[a] School of Computer Science and Technology in CUMT, Xuzhou, Jiangsu 221116, PR China
[b] Chinese University of Hong Kong, Shenzhen, Guangdong 518172, PR China

ARTICLE INFO

Article history:
Received 20 April 2023
Revised 11 June 2023
Accepted 30 June 2023
Available online 1 July 2023

Keywords:
Code reuse attacks
Operating systems
Software and system safety
Access control
Code probes

ABSTRACT

Due to the address space layout randomization (ASLR), code reuse attacks (CRAs) require memory probes to get available gadgets. Code reading is the basic way to obtain code information. In theory, setting the code to be unreadable can prevent code reading. However, the pages are loaded dynamically, and the existing methods cannot set all code as unreadable at one time. They can only control code permissions page-by-page via time-consuming page tracking. Moreover, since some special users need to read code, turning off the read permission will affect their execution. To solve these problems, this paper proposes a method AntiRead. It rebuilds the buddy system for memory allocation. The new buddy system places code pages in a specific memory pool to manage their read permissions. In the presence of AntiRead, what is obtained by adversaries through code reading is either randomized code or non-executable code. Experiments and analysis show that AntiRead can prevent the code that has been read from being used as gadgets without affecting the normal code reading. In addition, the CPU overhead introduced by AntiRead is 1.8%.

1. Introduction

CRAs is a control flow hijacking technology. It does not need to inject any code, but instead uses existing code snippets in the operating system (OS) as malicious payloads (called gadgets). Before an attack, the adversary must prepare all gadgets to build a gadget chain (Jang. 2022; Lu et al., 2021). The form of each gadget must conform to specific forms (such as *pop rax; jmp *rax*), and the address of each gadget must be known. Under the protection of ASLR, especially the fine-grained ASLR, the addresses and forms of code blocks are invisible to the adversaries. Therefore, they must perform code probing to obtain available gadgets (Lu et al., 2021).

Recent researches have demonstrated ASLR can be bypassed by exploiting a memory leakage vulnerability to harvest code pointers and disclose code memory on-the-fly (Zhang et al., 2017). Code reading is a basic method that is used by a variety of probing technologies. For example, attackers can obtain the library function addresses by reading PLT (procedure linkage table), which is essentially code reading (Hu et al., 2016).

The basic way to prevent code reading is the execute-only policy, which turns off the read permission of code. Such a policy can be implemented using page table manipulation (Backes et al., 2014), split TLBs (Gionta et al., 2015), hardware virtualization extensions (Crane et al., 2015; Werner et al., 2016), or a form of software-fault isolation (K et al., 2016; Pomonis et al., 2017).

However, the existing methods have some limitations. First, some methods rely on source code, which is invalid for the closed-source objects. For example, Readactor (Crane et al., 2015) needs to analyze source code to separate the mixed page containing code and data, which makes it unable to protect the loaded library code. Second, some methods introduce huge overhead to identify the physical pages used to store code and dynamically disable their reading permissions. For example, Heisenbyte (Tang et al., 2015) introduces over 60% overhead to *perlbench*. Third, almost all methods ignore the negative effects of turning off the read permission. The existing methods assume all users don't read the code. This is the basic premise that the application can still run normally after the code read permission is turned off. However, some special users, such as debuggers, require reading code. Although we can read the code of the target process in debug mode, this method is not applicable to all scenarios. For example, the code read in debug mode may be different from the code read in the scenario of re-randomization (Yun et al., 2020). The best way to solve this problem is to enable the read permission of the code pages when they are being read, and ensure the code snippets that have been read cannot be used as gadgets. Forth, some methods, such as NEAR (Werner et al., 2016), cannot protect the page, in which the code is swapped into memory again. That is, if a code page is loaded into memory again after being swapped to the disk, it may be loaded into a readable page, which leads to a protection failure.

* Corresponding author.
E-mail address: liyg@cumt.edu.cn (Y. Li).

https://doi.org/10.1016/j.cose.2023.103377
0167-4048/

One of the reasons causing the above problems is the existing methods cannot preset all binary code of the target process to be unreadable. In fact, it is not easy to achieve this purpose. Because the OS will not load all code into memory at once. Before the code is called for the first time, it is in the disk rather than the memory. Therefore, the code pages cannot be known or predicted in advance, which makes the permissions of the binary code cannot be preset. Now, there are two methods to solve this problem. One is to mark the target code via the compiler. Then, the code pages can be obtained when the code is called. Such a method needs to manipulate the source code and cannot protect the closed-source objects. The other is to track the page allocation and code accesses to identify code execution and code reading. Such a method requires frequent intervention in code execution, which will incur significant overhead.

Another reason for the above problems is directly disabling the read permission of code pages will cause some negative effects. In practice, both the debugger and the system-level optimizer need to read the code of applications and even the shared libraries. The unreadable code will lead to a failure.

To solve these problems, this paper proposes a novel method AntiRead. It can protect the closed-source objects including shared libraries. AntiRead combines virtualization technology to rebuild the memory allocation system *buddy system*, which can preset all code pages to be unreadable and avoid the huge overhead caused by tracking the accesses to code pages. It can also ensure the swapped code pages cannot be read after they are loaded into memory again. In the presence of AntiRead, the code that have been read can still be called but cannot be used as gadgets. In summary, the contributions are as follows:

(1) Propose a system to manage code pages. This method rebuilds the *buddy system*. It can set all the code pages to be unreadable before code loading, which reduces the overhead of tracking code accesses.
(2) Propose a mechanism to defend against malicious code reading. It can prevent the binary code that has been read from being used as gadgets. Meanwhile, the legal applications can read the code they need.
(3) Implement the protype of AntiRead in Linux. To the best of our knowledge, AntiRead is the first method that can preset all code pages as unreadable. It has good defense effect on the code probes based on code reading and only introduces 1.8% overhead to CPU.

2. Related works

The adversary can obtain the code address and code forms by reading code, which is a basic way to build a gadget chain (Li et al., 2022, Schloegel et al., 2021). In response to such probing attacks, researchers have proposed various security solutions, including ASLR and execute-only memory (XOM).

2.1. ASLR methods

ASLR scrambles the memory distribution, making the addresses of the target objects unknown to the adversary. As a result, the adversary cannot connect gadgets together, even if their forms are known. The existing objects protected by ASLR include code pages (Crane et al., 2016), functions (Conti et al., 2016), basic blocks (Wartell et al., 2012), and instructions (Hiser and Nguyen-Tuong, 2012). ASLR has been widely used in the OS.

However, the calling relationship between code objects will become more and more complex as the code size increases. Randomizing the whole code segment or the entire memory object requires modifying the instruction paths between all code objects to maintain the original execution logic, which is time-consuming. Moreover, fine-grained randomization increases the operation complexity. To accurately profile the calling relationship between code objects, many ASLR methods have to analyze the source code. Remix (Chen et al., 2016) is a LLVM-based method and it adds extra *nop* paddings to change the code addresses, which allows runtime flexibility for moving code inside functions. TASR modifies GCC and the dynamic linker to rerandomize the memory layout during runtime before the adversary can take advantage of any stolen knowledge (Bigelow et al., 2015).

Adelie is a kernel ASLR method and proposes the mechanisms for stack re-randomization, address encryption, and continuous ASLR on Linux modules (Nikolaev and Nadeem, 2022). Shuffler is a runtime ASLR solution and it randomizes code asynchronously in a separate thread (Williams-King et al., 2016). CodeArmor virtualizes the code space to completely decouple code pointer values from the concrete location of their targets in the memory address space (Chen et al., 2017). PT-Rand randomizes the location of page tables and tackles several challenges to ensure that the location of page tables is not leaked(Davi et al., 2017). The main idea behind ASLR-GUARD is to render leak of data pointer useless in deriving code address by separating code and data, provide a secure storage for code pointers, and encode the code pointers when they are treated as data (Lu et al., 2015). CoDaRR continuously rerandomizes the masks used in load operations and storage operations, and re-masks all the related memory objects, which can maintain the transparency of code execution (Rajasekaran et al., 2020).

However, it turns out that the ASLR can be bypassed by code probing technologies such as JIT-ROP (Ahmed et al., 2020). Because there is no ASLR that can completely hide all the code and code pointers. An adversary can still obtain the addresses and forms of the target code directly or indirectly. In addition, almost all ASLR methods randomize the whole code segment or the entire memory object, such as (Backes and Nürnberger, 2014; Sun and Lui, 2016; Giuffrida et al., 2012), which is complex and time-consuming. In practice, under the protection of fine-grained ASLR, adversaries cannot speculate all addresses through a single leak or probe. Therefore, most code information is still unknown to adversaries. That is, there is no need to randomize all the code.

AntiRead also adopts a fine-grained ASLR method. Unlike existing methods, AntiRead only randomizes the code page that is being read, not the whole code segment or the entire memory object. Moreover, AntiRead does not require the randomized code to be executable. Based on the above design principles, it does not need to maintain the complex calling relationship between code objects, which is the key to have high efficiency.

2.2. XOM methods

Existing XOM methods disable the read permissions of code pages, thereby preventing code information leakage. XnR ensures the code can still be executed by the processor, but it cannot be read as data (Backes et al., 2014). HideM uses the split-TLB architecture, commonly found in CPUs, to enable fine-grained execution and read permission on memory (Gionta et al., 2015). NORAX leverages a combination of MMU permission bits to retrofit XOM to ARM binaries (Chen et al., 2017). In contrast, kRX enforces XOM on architectures that lack native support for marking memory pages as execute-only and employs strong memory isolation mechanisms, avoiding the use of information hiding to guard against JIT-ROP attacks (Ahmed et al., 2020). Central to Heisenbyte is the concept of destructive code reading – code is garbled right after it is read (Tang et al., 2015). Readactor (Crane et al., 2015) protects both statically and dynamically generated code. It uses a compiler-based code generation paradigm that uses hardware features provided by

modern CPUs to enable XOM and hide code pointers from leakage to the adversary. NEAR (Werner et al., 2016) foregoes the problems of XnR (Backes et al., 2014) and provides strong security guarantees against just-in time attacks in commodity binaries.

However, the existing methods have some limitations, making them difficult to be widely adopted. XnR is susceptible to disclosure attacks via indirect code references. CodeArmor are ineffective against brute force attacks, such as code reading via HeartBleed (Li and Guoyuan, 2023). Readactor relies on source code. Heisenbyte has no protection effect on shared libraries. More seriously, all methods that prohibit code reading ignore the negative effects caused by closing the read permission of the code. Moreover, some methods introduce significant overhead when tracking code access.

Compared with existing methods, AntiRead does not completely disable the read permission of code. Therefore, the applications that need to read code can still get what they want. Meanwhile, AntiRead can prevent the code that has been read from being used as gadgets, which is a capability that existing ASLR methos do not have.

3. Assumptions and threat models

First, we assume the fine-grained ASLR is in use, and adversaries cannot infer the locations of all gadgets from a leaked code pointer. In practice, the operation granularities of the existing ASLR methods include pages, functions, basic blocks and instructions. Second, we assume adversaries can continuously probe the code without worrying about the interruption of code probes caused by process crash. For the code-reading-based probe, it may trigger exceptions when reading the unmapped area, which leads to process crash. While, the OS allows applications to handle exceptions by themselves to avoid process crash. Third, we assume adversaries can hijack the control flow by modifying return addresses or function pointers through memory vulnerabilities. Fourth, we assume adversaries cannot obtain memory layout through */proc* files.

There are three types of probing attacks based on code reading. The 3 attack vectors are as follows:

Vector 1. It gradually moves from the leaked data segment to the code segment. HeartBleed (Zhang et al., 2014), a classic attack, can disclose 64KB data at a time. By correcting the data pointer multiple times, the target area can gradually approach the code segment. The adversary can identify the real code segment by checking the binary forms (such as the most common function header *55 48 89 e5*). Although an adversary may read unmapped areas, this does not prevent code reading activities. Because the adversary can restart the process, and can also handle the error signal SIGSGEV itself.

Vector 2. It directly reads the code via the leaked code pointer, and recursively read the code in the mapped area pointed by indirect addresses in code pages. The typical representative of this attack is JIT-ROP (Ahmed et al., 2020). Since dynamically compiled code is used, all compiler-based methods are invalid for such attacks.

Vector 3. It obtains the target addresses by reading the code pointers stored in the continuous memory. The typical attack is data leakage (Liljestrand et al., 2019). It obtains the GOT (global offset table) address where stores all library function pointers needed by the current process through the relative address stored in PLT.

3.1. Overall design

In the original OS, the code is readable. To defend against the probes based on code reading, the existing mthods directly disabling the read permission of the code. The existing memory system does not distinguish which pages are used to store code and

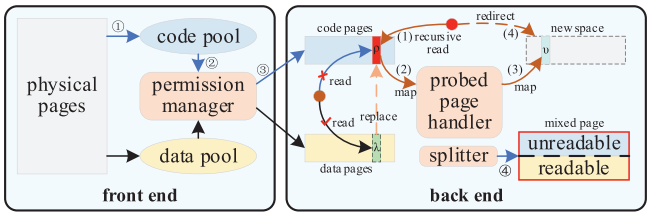

Fig. 1. Overall architecture of AntiRead.

which pages are used to store data. In addition, all code pages of the process are not allocated at once. Therefore, before the code is executed, we do not know which page needs to be set as unreadable. Moreover, the OS cannot directly disable read permissions of memory pages. To identify each code page, all page allocation needs to be tracked. To disable the read permission of the code pages, all code access needs to be detected. Such designs trigger significant runtime delays. In addition, disabling the read permission of all code pages will affect the normal code reading activities.

AntiRead can solve the above problems, as shown in Fig. 1. It consists of two parts, the *front end* and the *back end*. The *front end* works offline and has the component *permission manager*. The *back end* works online and includes the components *splitter* and *probed page handler*. The *front end* provides a *code pool* and a *data pool* for the *back end*. The *back end* allocates memory for the process according to the execution scenarios, and ensures the code that has been read can't be used by CRAs.

The *permission manager* reconstructs the memory allocation system *buddy system*. It places physical pages in the *code pool* and *data pool*, and sets different permissions for them. Then, the user code pages can only be allocated from the unreadable *code pool* (①~③), and the data pages and kernel pages can only be allocated from the readable *data pool*. The *permission manager* does not introduce huge overhead like XnR (up to 526% in Jan's test (Werner et al., 2016)) because it does not need to track every code page allocation and code access on-the-fly.

A mixed page contains both code and data. If the page is directly set as unreadable, the data in the page will not be able to be read normally. To map the code and data in the mixed page to unreadable and readable separately, the code and data need to be stored in different pages. When the process is loaded, the mixed page is identified by the *splitter*. *Splitter* migrates the code in the mixed page to a new space, which can place the code and data in separate pages. The new code page is executable but unreadable, while the original page is readable but non-executable (④).

The idea behind *probed code handler* is to disable the execution permission of the target code (instead of the code segment) and randomize its memory layout when it is being read. Therefore, the attacker cannot execute the detected code, even if it already knows the code address. In addition, *splitter* also ensures that the code that has been read can still be legally called. To achieve the above purposes, we replace the code page (ρ) that is being read with a readable but non-executable code page (λ). Meanwhile, the code page that is being read will be mapped to an executable page (υ) in a new address space. Afterwards, all control flows that jump to λ will be captured and analyzed. All illegal control flows will be blocked, while the legal control flows will be redirected to υ.

It should be noted that the code in υ will be copied into λ. Then, the code in λ will be randomized at function and code block granularities. All the relative addresses of the jump instructions (such as *call/jmp address*) in λ are modified to make them point to the new address space. Therefore, if an adversary reads code pages recursively starting from the probed page like JIT-ROP, it will read the contents of the new address space, as shown in (1)~(4).

For read requests in the new address space, AntiRead copies each code page that is being read and sets it to be readable but non-executable. Therefore, the code obtained by the adversary is either randomized or non-executable.

The above designs require the capabilities of monitoring, tracking and controlling the OS resources and the process behaviors. To achieve this purpose, we combine VMX (Virtual Machine Extension) root and VMX non-root to divide the running modes of the original OS into two types, *host* and *guest* (Li and Chung, 2022). In a general scenario, the OS runs in the *guest*. When a specific event occurs, the running mode of the OS will switch from *guest* to *host*, which is called a system trap. Combined with EPT (Extended Page Tables) and VMX, AntiRead can set various system trap events including process switching, the execution of specific instructions (such as *int3*), interrupts, and debug exceptions, etc. During handling system traps, the process is suspended, and its resources and state can be detected and modified. After that, AntiRead can control the execution of the process by modifying the VMCS (virtual machine control structures) fields. The control events include injecting general protection exceptions, setting breakpoints, modifying CPU context, and redirecting control flow. For example, after modifying the *guest rip* in VMCS, the control flow can be redirected. In summary, AntiRead can monitor, track and control the execution of processes.

4. Implementation of AntiRead

Next, we introduce the implementation of *permission manager*, *splitter*, and *probed code handler* in detail.

4.1. Permission manager

The existing XOM methods require tracking page allocation and code access to disable the read permission of code, which is a key factor that incurs overhead. If all code pages of the target object can be set as unreadable in advance, runtime overhead will be reduced. To achieve this goal, the *permission manager* combines with EPT technology to reconstruct the memory allocation system *buddy system*, as shown in Fig. 2.

In Linux, the code page allocation in user space is done by the *buddy system*. Physical memory is organized by the *zone list, zone* and *page*. In NUMA architecture, the *buddy system* uses two *zone lists* for each node to manage all memory *zones*. The first *zone list* is used to manage the *zones* directly connected to the current CPU, and the second *zone list* is used to manage all *zones*. When a page fault occurs, the *buddy system* will select a specific number of pages from a *zone* in the two zone lists (the first is preferred) for the process. The original *buddy system* does not differentiate between code pages and data pages, which poses a challenge for identifying code pages.

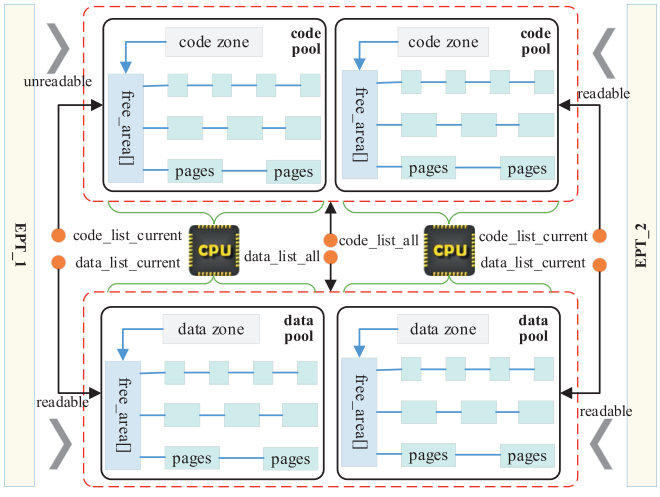

Fig. 2. New buddy system.

Compared with the original *buddy system, permission manager* doubles the number of *zone lists* and *zones*. In the new *buddy system*, all the memory directly connected to the current CPU is divided into two categories. One is used to allocate for user code, called *code pool*; the other is used to allocate for other objects other than user code (such as kernel code and user data, etc.), called *data pool*. The *zones* contained in the *code pool* are called *code zones*, and the *zones* in the *data pool* are called *data zones*. *code_list_current* points to the *code zones* that directly connected to the current CPU, and *code_list_all* points to all *code zones*, including the *code zones* connected to other CPUs. *data_list_current* points to *data zones* directly connected to the current CPU, *data_list_all* points to all *data zones*, including the *data zones* connected to other CPUs. Based on the above designers, code pages and data pages in user space can be separated.

Next, code pages and data pages can be pre-set with different permissions. We respectively set *code zones* and *data zones* to be unreadable and readable via EPT_1. To prevent *code zones* from being tampered with, we also set them to be unwritable. When a page fault occurs, the modified function *__alloc_pages_nodemask* (a kernel function used to allocate pages) allocates pages from *code pool* or *data pool* according to the fault type. The fault type (code page fault or data page fault) can be determined by checking wether the fault address stored in the register *cr2* points to code areas.

It should be noted that the code pages cannot be set to be both unreadable and writable at the same time. Otherwise, it will cause an EPT misconfiguration. When loading code, the code page must be writable. Otherwise, an EPT exception will be triggered. Therefore, we cannot complete code loading with EPT_1 due to the unreadable and unwritable *code pool*. To solve this problem, we must set the code pages as writable and readable during code loading. If and only if a code page fault occurs in user space, we execute the VMX instruction *vmfunc* at the head of the function *filemap_fault* (a function that moves code from an ELF file into memory) to switch EPT to EPT_2. At this point, the code page is readable and writable. When *filemap_fault* returns, *vmfunc* is executed again to switch EPT back to EPT_1. At this point, the code page has been loaded into memory, and the code page is executable but unreadable and unwritable.

However, JIT applications do not use *filemap_fault* to load code into the solidified code segment. It loads code into a code cache, which is essentially an executable heap. The OS still uses *alloc_pages_nodemask* to allocate memory for the code cache. Therefore, the physical page obtained by JIT is executable but unreadable and unwritable. When it loads code into the page, an EPT exception is triggered. After that we switch the EPT to EPT_2. It should be noted that the entire code cache will be set as non-executable in EPT_2. Therefore, executing any code in the code cache will cause an EPT exception, which means that the code has been loaded into memory at this time. After that, we switch the EPT back to EPT_1. Unlike loading pre-compiled code, JIT introduces two system traps between loading the code and executing the code.

When the code page that has been swapped out is called again, the kernel function *do_swap_page* will allocate a new physical page for it. This page may be in the assigned *swapper_space*, or it may come from a page allocated by the function *_get_free_page*. To ensure the replaced page is executable but unreadable, we detect whether the allocated page comes from *code pool* before the *do_swap_page* returns. If not, we set the page to be executable but

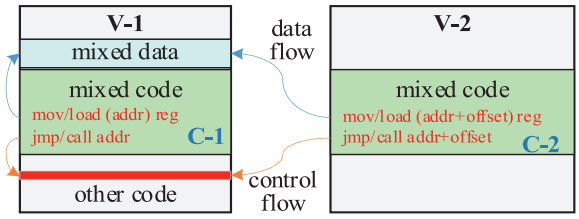

Fig. 3. The separation mechanism of mixed page.

unreadable. Therefore, the pages that are swapped in again are still unreadable.

We also create a set of remapping tables for the OS through Intel VT-d, which will mask the physical memory in the *code pool*. As a result, an adversary cannot read user code through DMA (direct memory access).

We divide all pages into two parts, which may cause the problem of unbalanced memory between the *code pool* and *data pool*. For example, there are many free pages left in the *code pool*, while there are few free pages left in the *data pool*. To solve this problem, the numbers of pages contained in the *code pool* and *data pool* are not absolutely fixed. When there are no enough free pages, the *buddy system* calls *__alloc_pages_slowpath* to free some pages that have not been recalled. If the current allocation requirements cannot be met after *__alloc_pages_slowpath* returns, it indicates that there are few free pages left in the current pool. When the *nopage* in *__alloc_pages_slowpath* is executed, we extract half of the free pages in another pool and migrate them to the current pool.

In summary, when a code reading occurs, the activity will trigger a system trap, and it will be captured. Moreover, the EPT switch caused by *vmfunc* does not cause a system trap, and the overhead it introduces is significantly smaller than handling a system trap.

4.2. Splitter

The idea behind the *Splitter* is to store the mixed code and data in different pages and give them different permissions, as shown in in Fig. 3.

First, *splitter* needs to distinguish between code and data in the mixed page. The mixed page is in executable code pages. The data in the mixed page includes *ELF headers* and *.rodata*, and they can be identified by analyzing the sections stored in ELF files.

Sceond, the code (C-1) in the mixed page will be migrated from the original space (V-1) to a new space (V-2). The original mixed page will be set as readable and non-executable. Therefore, the data in this page can be accessed normally, while the code cannot be called. To prevent the readable code in the mixed page from leaking memory layout, C-1 will be randomized w at the granularity of code blocks.

Third, *splitter* should ensure the legal control flow and data access can reach the right locations. When C-1 is called, an exception is triggered. After that, the control flow will be redirected to the migrated code C-2 in V-2 by setting the *guest rip* in VMCS. If the excepted instruction is *call/jmp addr*, the operand *addr* will be fixed to point to C-2. When it is called again, the control flow directly jumps to C-2 without triggering an exception. If the jump target of the instruction *call/jmp addr* in V-2 is not in C-2, we also modify its operand so that it can directly jump to the right code. Since *call/jmp *reg/pointer* in C-2 uses an absolute address, it can directly jump back to V-1. All data accesses in V-2 can be captured due to triggering permission exceptions. After that, they will be redirected to V-1.

To sum up, *splitter* can ensure the data in the mixed page can be accessed normally. Meanwhile, the code in V-1 is obfuscated, and the code in V-2 is unreadable, which can prevent code information leakage.

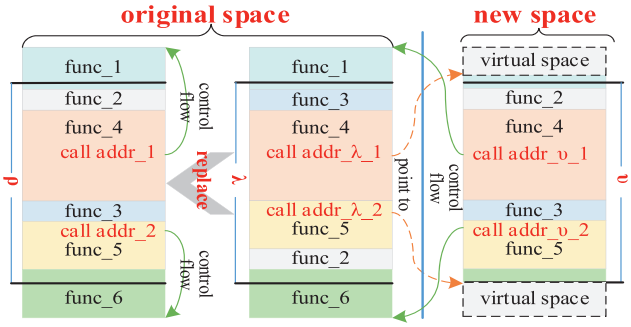

Fig. 4. The implementation of page code handler.

4.3. Probed code handler

Although the existing XOM methods can prevent code information from being read, it also stops the legal code reading. For the code that is being called, we can directly disable its execution permission, which can prevent the leaked code from being used as gadgets. However, such a design also prevents the protected code page from being called legally. Especially for multithreaded scenarios, the code that is being read may is being called by other threads.

To solve the above problems, we should ensure the code can be read nomally. Meanwhile, the code that has been read cannot be used as a gadget, while it can be called legally. The main idea behind the *probed code handler* is to disable the execution permission of the code page that is being read in the original space and migrate it to an executable page in a new space. The implementation of *probed code handler* is shown in Fig. 4.

The original code page υ that is being read will be replaced by a non-executable and randomized code page λ, as shown in Algorithm 1. Then, υ will be mapped to the new space (line 4). When the original code that has been read is called again, the control flow will be transferred to the non-executable λ, which trig-

Algorithm 1
The method handling probed code page.

Input: ρ, the probed code page; *Ins*, the excepted instruction; *func*, the function reading code; *ret_addr*, the return addresses on stack.
Output: NULL
1. **If** abnormal_access(ρ)=1 **then** // find the code reading
2. ψ=Build_new_space() // build a new address space ψ
3. $\upsilon \in \psi$ // the new page is in ψ
4. Exchange_mmap(ρ, υ, ψ) // map ρ to υ in new space ψ
5. λ=allocate_data_page(sizeof(ρ)); // allocate a data page λ
6. Code_copy(ρ, λ) // copy the code of ρ into λ
7. Reorder(λ) // Randomize the memory layout of λ
8. **Foreach** *indirect_address*$\in\lambda$ **do**
9. Redirect(*indirect_address*, ψ) //redirect *indirect_ address* to ψ
10. **End Foreach**
11. RedirectCF(*Ins*, υ) // redirect the control flow to υ
12. **Foreach** *indirect_address* $\in\upsilon$ **do**
13. **If** *jmp/call indirect_address triggers a system trap* **then**
14. RedirectCF (*jmp/call indirect_address*, *OrigSpace*)
15. **End If**
16. **End Foreach**
17. **End If**
18. **If** $\lambda\leftarrow$*Ins* **then** // the control flow jumps to λ and causes exceptions
19. RedirectCF(*Ins*, υ) // redirect the control flow to υ
20. **End If**
21. **If** $\nexists$ rip $\in$ {*func*, υ} && $\nexists$ *ret_addr* $\in$ {*func*, υ} **then**
22. Exchange_mmap(υ, ρ, *OrigSpace*) // map υ to ρ in *OrigSpace*
23. **End If**

gers a system trap. Next, we redirect the control flow from λ to υ by modifying the field *guest rip* in VMCS (line 11). Based on this design, we not only ensure the code can be read normally, but also ensure the code that has been read cannot be used as a gadget.

The new virtual space is the same size as the original space, and it also contains the virtual spaces of code segments and data segments. In the new space, except υ, other virtual pages in the code segment will be redirected to a page that is unreadable, unwritable and non-executable. Therefore, the control flow can only be transferred to υ, otherwise an EPT exception will be triggered. The virtual pages corresponding to the data segment in the new space will be mapped to the real physical data pages corresponding to the original space, which ensures the code in υ can access the original data.

λ is allocated to replace ρ, which can be achieved by modifying the item in the last-level page tables of the EPT. λ and ρ include the same code. While, the memory layout of the functions and the code blocks in λ is randomized (line 7). After that, we modify the operands of the current instruction that attempts to read code to ensure the right code can still be read. The code block we select for randomization uses *jmp xxxx* or *ret* as an exit, whose next instruction is antother code block's entry. Randomizing such code blocks will not affect the execution logic of the code. Because they have no return relationship with their adjacent code blocks like *call XXXX*, nor are they affected by the execution conditions like *jne*.

Considering some attacks (such as JIT-ROP) can read more code pages recursively through indirect addresses stored in λ (such as the *address* in *call address*), we fix the indirect addresses contained in λ. All indirect addresses in λ will be modified to make them point to the new address space (lines 8-10). When an adversary recursively reads the code page with the modified *address*, it will be captured due to EPT exceptions. After that, we copy the code page to be read into another readable but non-executable data page λ-1 that is in the new space. The relative *address* in λ-1 points to the code in the new space. Therefore, we need not to modify its indirect addresses. When λ-1 is read by adversaries, what they obtain are in non-executable pages. In this kind of push, we will allocate readable but non-executable pages (λ-2, λ-3, ..., λ-n) for the objects with code reading needs until the end of the reading activities. This design can prevent an adversary from building a gadget chain. For the applications that need to read the code, they can still read the right code. Although the relative address of the code has been changed, this does not have any impact on the execution logic of the code.

To ensure the code in the new space can call other code in the original space, we should redirect the control flow to the right location. For the *jmp/call address* in υ, when it triggers a system trap, we directly redirect the control flow to the original space by modifying the *guest rip* in VMCS (lines 12-16). For the instruction *jmp/call *reg/pointer* that use absolute addresses, they can jump directly back to the original address without any corrections.

Each control flow transfer instruction jumping to λ will trigger a system trap, which introduces significant overhead. To reduce the overhead, we map the υ in the new space back to the original space after the read activity is completed (lines 21-23). Since the content read by an attacker is either randomized code or non-executable code, the code snippets in υ cannot be used as gadgets, even they are mapped back to the original space.

For the shared library, its memory layout in the current process is the same as that in other processes. Even if their base addresses may be different, the offset between two different base addresses can be inferred from the leaked pointers, such as the return addresses. Then the base address can be calculated based on the offset. As a result, the memory layout of the shared library can be used in different processes once it has been known by attackers. To make matters worse, the new space used by the above design

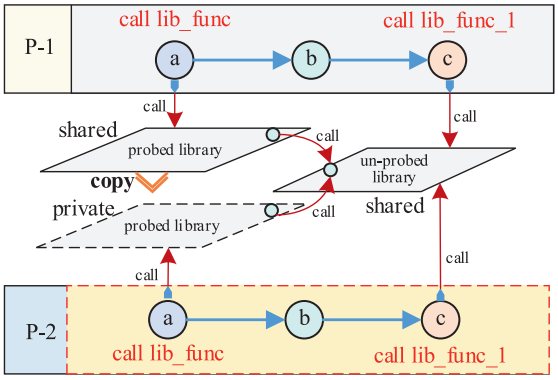

Fig. 5. The implementation of page code handler.

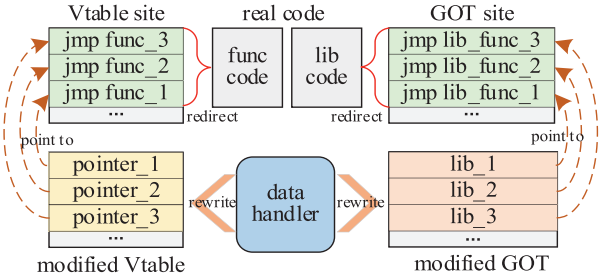

Fig. 6. The mechanism of data redirection.

will affect the normal execution of the shared code in the parallel processes. The reason is the new space is only mapped to the address space of the current process, not all processes. Therefore, the control flow of other processes cannot be transferred into the new space. In addition, the direct manipulation to shared library may cause execution conflicts, unless all processes calling the shared code are suspended. The idea to solve these problems is to convert the shared library code that is being read into the private code of the current process, as shown in Fig. 5.

The shared code page that is being read will be copied to a new physical page. Next, the last level of the page table is modified to map the virtual address of the shared code page to the new physical page. Then, the target to be protected becomes a private code page, which does not affect other processes calling the original library code. As a result, we can manipulate the private library code as it handles the application code. From the attacker's perspective, the code he reads is randomized code, which is different from the code layout in the address space of other processes.

In the presence of fine-grained ASLR, an adversary cannot infer the memory layout of the entire process through a single leaked function pointer. Unfortunately, in addition to the indirect addresses in the code, Vtable (virtual table) and GOT also contain rich address information. Once they are read by adversaries, many function pointers will be disclosed.

To prevent adversaries from getting addresses in Vtable and GOT, we build a data redirection mechanism, as shown in Fig. 6. Vtables can be identified by scanning the content in *.rodata*. In contrast, the GOT is stored in the writable area and its address (relative address) is stored in PLT in the code segment. An EPT exception is triggered when an adversary reads PLT. After that, we can speculate the adversary may have the intention to probe the GOT.

The data in Vtable will be rewritten when the process code is loaded into memory. The data in GOT will be rewritten when PLT is read. After data rewriting, they respectively point to *Vtable site* and *GOT site* instead of the original targets. Both *Vtable site* and

GOT site are executable but unreadable. What in *Vtable site* and *GOT site* is only the transfer code that can redirect control flow to the target code. When adversaries get the addresses in Vtable or GOT, what they get is just the address of transfer code, not the real address of the target function. Therefore, an adversary cannot infer the gadgets contained in the function based on the first address of the probed pointer.

Unlike Vtable, the entries in GOT are not loaded into memory at once. Therefore, the function pointers in GOT cannot be rewritten all at once. We can only modify the function pointer written into GOT one by one. After PLT is read, GOT will be set as unwritable to capture the entry to be written. When an EPT exception occurs in GOT, GOT will be adjusted to be writable, and the single-step debug mode is enabled by setting the registers *dr0*∼*dr7*. After the entry is written into GOT, a system trap is triggered. Next, the entry will be modified to point to the *GOT site*. Finally, the single-step debug mode is canceled, and the write permission of GOT also be canceled. In this way, every function pointer written into GOT can be hidden before it can be read.

5. Evaluation

We conduct all experiments on a Linux server, which is equipped with two 10-core Intel Xeon silver CPUs and 128GB memory. The OS is Ubuntu18.04 with kernel 4.16. It should be noted that all performance evaluation results are the average of 10 runs.

5.1. Security evaluation

The protection effect on Vector 1. The adversary can start from the data area and gradually move closer to the code segment until the code is read. During this period, the adversary may read the unmapped area, which triggers the signal SIGSGEV and causes process crash. However, the adversary can perform the next round of code probing after restarting the process. We deploy such an attack HeartBleed in *openssl-1.0.1c* to simulate Vector 1. To read the process code, the parameter *pl* of the *memcpy (bp, pl, payload)* in *openssl* gradually decreases. This operation can extend the leaked content from the data area to the code area. When the signal SIGSEGV is triggered, the current process will be restarted for the next round of code probing. In our test, HeartBleed read the first code page after about 1200 probes. We found the code read by HeartBleed is not in the same order as the code in the original code page. The indirect addresses (such as the *address* in *call address*) are also not the same as the addresses contained in the executable code pages.

The protection effect on Vector 2. The adversary can also use the leaked code pointers to read the code directly. Under the AntiRead's protection, no matter how the adversary reads the code page, the code he can get is either re-randomized or non-executable. Because an EPT exception will be triggered due to the code reading. After that, the code pages that are being read will be mapped to a new space. Meanwhile, the pages an adversary can read will be re-randomized, and the indirect addresses in the page will be modified. As a result, what adversaries read cannot be used as gadgets. For the indirect addresses contained in the code page that has been read, they no longer point to the executable code pages, which makes adversaries, such as JIT-ROP, unable to recursively read the executable code.

To observe the changes of code layout and code forms before and after code reading, we use a loadable kernel module (LKM) to read the code of *perlbench* in SpecCPU2006, as shown in Fig. 7. The result shows that the layout of the code that has been read is different from the layout of the original code. In fact, both the base address of the function and the relative position of the code block inside the function have changed. For example, after the code at $0 \times 40394b$ is read, its address becomes $0 \times 403e24$; the distance between the original code block and the function head is $0 \times 3db$, while the distance between the randomized code block and the function head is 0×504. After that, we read the code at 0xff59ba60 through the indirect address stored in [$0 \times 403e35$∼$0 \times 403e38$]. What we read in 0xff59ba60 is the same as the code in the original address $0 \times 48aa60$. However, the code in 0xff59ba60 is non-executable. In summary, for Vector 1 and Vector 2, the code they can read is either randomized or non-executable. As a result, what the attackers obtain cannot be used as gadgets for CRAs. Even the CRAs with complete functions as gadgets cannot be deployed.

The protection effect on Vector 3. For the code pointers stored in GOT and Vtable, they have been modified to point to the transfer site instead of the original functions. Therefore, adversaries cannot guess the real code snippets in the target functions based on what they read.

The protection effect on the cloned process. The existing attacks can obtain child processes with the same address space as the parent process through process clone. The code information obtained by the adversary from the child process can be used to build a gadget chain in the parent process without causing a crash of the parent process. Under the protection of AntiRead, the code cloned by the child process is still unreadable. For the code pages that have been read in the child process, they will be re-randomized. Therefore, the code layout obtained by reading the child process code is different from the layout of the code in the parent process. Although adversaries can obtain non-randomized code through Vector 2, the code is stored in non-executable pages. As a result, an adversary cannot build a gadget chain based on the cloned process.

In summary, the users with code reading requirements (such as debuggers) can still obtain the code with normal logic. Although such code can also be obtained by attackers, they cannot be used to form a gadget chain. To verify this conclusion, we use an LKM to read the code in *perlbench* and *gcc*, and observe the jump numbers of the illegal control flow in the code that has been read and randomized. The results are shown in Fig. 8.

Before the test, we use ROPgadgets [42] to search for available gadgets in the binary code of *perlbench* and *gcc*. They contain 100750 and 254156 gadgets, respectively. Then, we randomly select 200 gadgets from each gadget group. Next, we use LKM to read the code page containing the selected gadgets to trigger the code randomization done by AntiRead. Finally, we redirect the control flow to the selected gadgets by manually modifying the return addresses, and observe the jump numbers of illegal control flow through LBR (Last Branch Record) register group.

The results show that the illegal control flow can only jump 5 times in the randomized code at most. In most cases, the jump number of illegal control flow is less than or equal to 1. Therefore, we believe that even though attackers can still get the right code forms, they cannot build a complete gadget chain.

5.2. Performance evaluation

We use SpecCPU2006 to meature the CPU overhead introduced by AntiRead, as shown in Fig. 9. Meanwhile, we use an LKM to read code pages with different proportions from high address to low address, which simulates Vector1. The results show that when there is no code reading, AntiRead introduces an average of 1.8% overhead to the CPU. When all the code is read, AntiRead introduces about 48.4% CPU overhead on average.

We also use the indirect address in the code page (such as the *address* in *call address*) to recursively read the code pages with different proportions, which simulates Vector 2. In this scenario, the

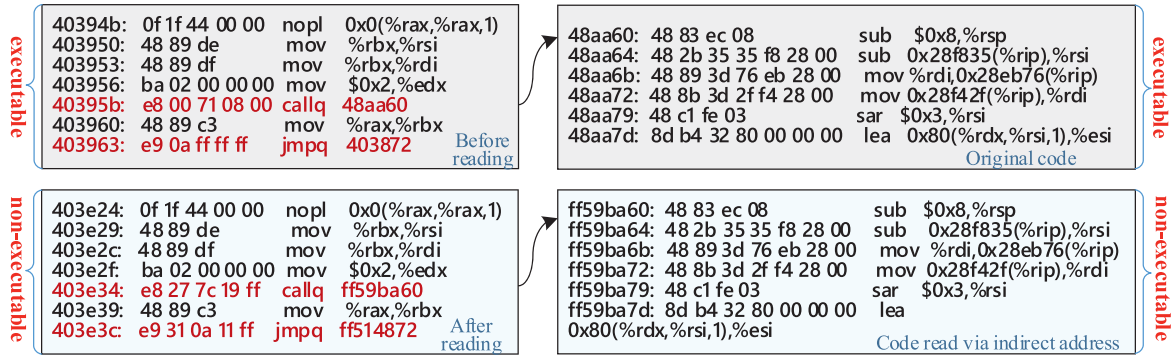

Fig. 7. The code changes in permissions, memory layout, and forms after code reading occurs.

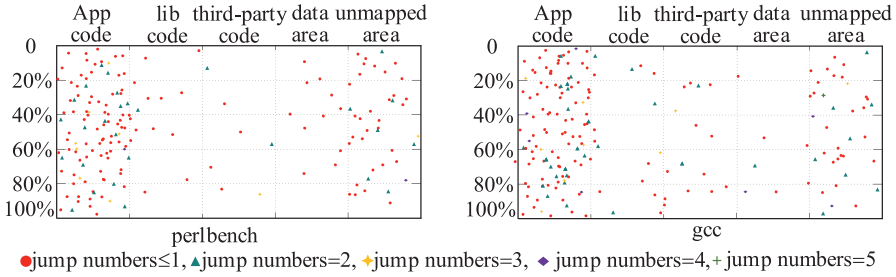

Fig. 8. The jump numbers of the illegal control flow in the randomized code. The ordinate indicates that the illegal jump is initiated at x% of the address space in perlbench or gcc, and the abscissa indicates that the illegal jump ends in the target object. In the abscissa, each line segment corresponds to the whole address space of a target object. The leftmost end of the line segment is the lowest address and the rightmost end is the highest address. Each marked point indicates that an execution error is triggered after n illegal jumps.

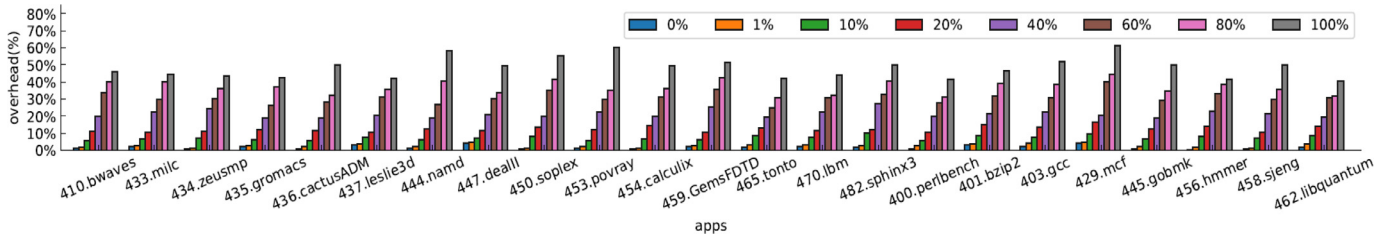

Fig. 9. The overhead measured by SpecCPU2006 when handling the code reading based on HearBleed.

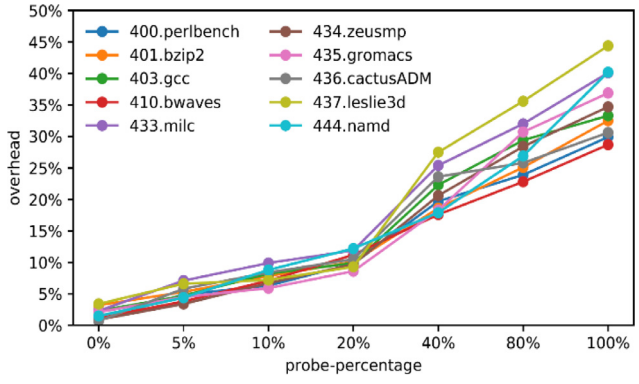

Fig. 10. The overhead of handling code reading based on JIT-ROP.

impact of AntiRead on the CPU is shown in Fig. 10. The results show that the overhead introduced by AntiRead will gradually increase with the size of the code being read. However, compared with handling the code reading based on Vector 1, AntiRead has less overhead in handling the code reading based on Vector 2. For example, when handling 20% of the code, AntiRead introduces an average of 12.4 and 10.3% CPU overhead in the two execution scenarios, respectively.

When handling Vector 1, AntiRead should copy the code, adjust permissions, randomize the code and modify the indirect addresses in the code page. After the code reading ends, the original code page should be restored. In contrast, AntiRead only needs to perform the above operations on the first code page being read when handling Vector 2. For the code pages that are read recursively, AntiRead only needs to copy them to the new space without other operations. Therefore, AntiRead can handle Vector 2 faster.

Although AntiRead introduces significant overhead when handling Vector 1, this overhead is not permanent. To verify this conclusion, we use web applications to measure the impact of AntiRead on their running speed before and after code reading, as shown in Fig. 11. For the web servers, the number of work processes is 4, the number of connections is 8, and the size of the requested file is increasing. We use an LKM to read different proportions of code from high address to low address in continuous virtual memory. During code reading, we measure the data transfer speed of the web applications. After 5 minutes, we measure the data transfer speed again. The results show that AntiRead significantly slow down the speed of applications when handling the code that is being read. After AntiRead finishes the code handling, this impact will become smaller, and even be equal to the impact

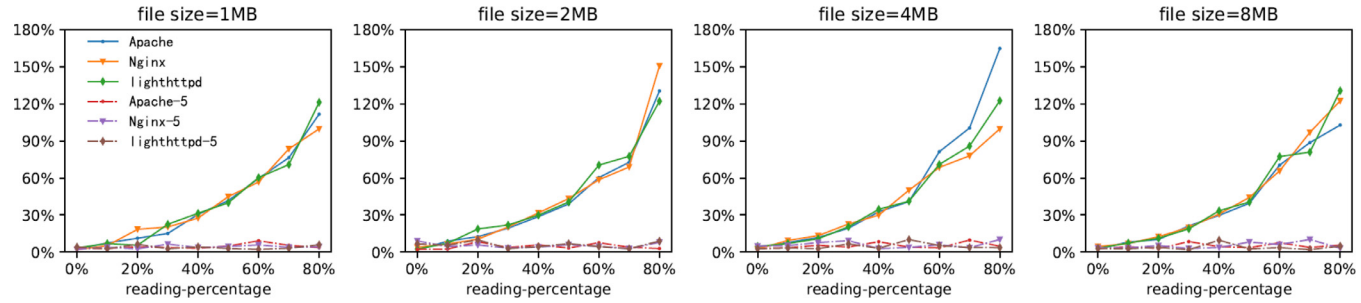

Fig. 11. Ovehead measured during and after handling the code reading. AppName-5: Overhead measured at 5th minute after the code is read.

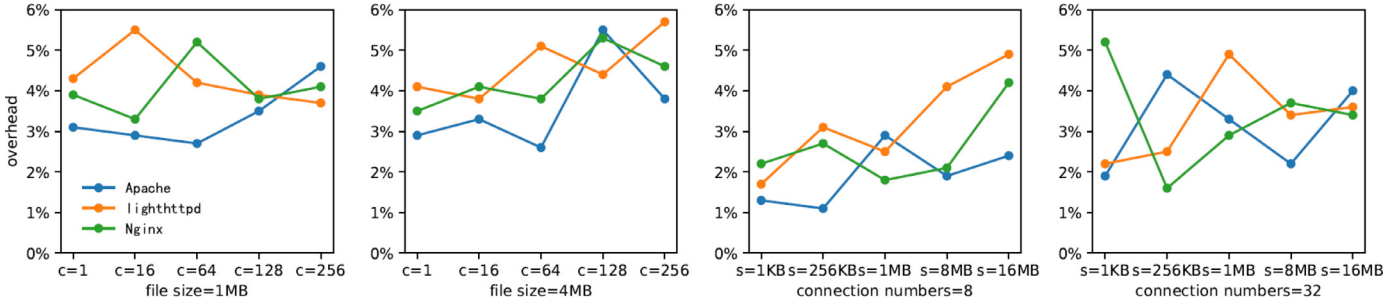

Fig. 12. Web overhead in normal scenarios. c: connection numbers, s: file size.

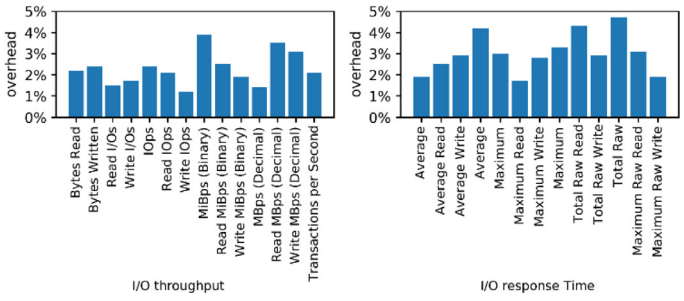

Fig. 13. The overhead measured by IOMeter.

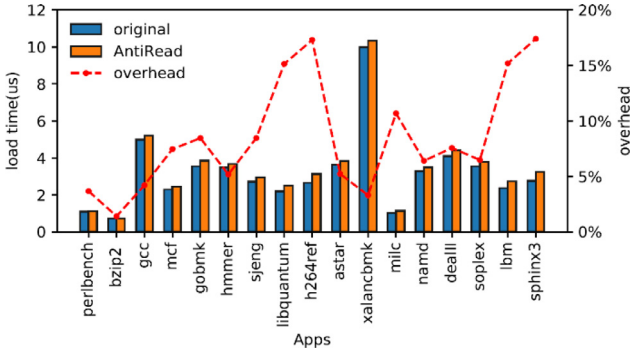

Fig. 14. The impact on the time of loading process.

before handling the code. The reason is that AntiRead will map the code that has been read back to the original address space after the code reading is finished. After that, the control flow can be directly transfered to the original code without triggering any system trap. Therefore, we do not need to track and redirect the control flows, which is the same as executing the code that is not read.

Although in the extreme cases, AntiRead will slow down the running speed of web applications by times, this does not mean it has a significant impact on network. In fact, code reading activities only occur in specific scenarios. Most of the time, the code in the application and library will not be read. To measure the impact of the AntiRead on network in normal scenarios, we measure the running speed of the web applications when there is no code reading, as shown in Fig. 12. For the web servers, the number of work processes is 4. The results indicate the average overhead on the network is about 3.3%.

We use IOMeter to measure the impact of AntiRead on I/O, as shown in Fig. 13. The results show that the I/O throughput is reduced by 2.3%, the I/O response time is increased by 2.9% on average.

When the process starts, AntiRead uses the new *buddy system* to allocate code pages for it. The code loading will trigger the EPT switch, thus increasing the time of loading process, as shown in Fig. 14. The results show that AntiRead increases the load time by 2–18%.

Table 1
Jit V8 test.

Benchmark	Orig.	AntiRead	Overhead
Richards	38421	37257	3.03%
DeltaBlue	59788	57174	4.37%
Crypto	32006	30398	5.02%
RayTrace	78125	74695	4.39%
EarleyBoyer	43076	41992	2.52%
RegExp	5937	5621	5.32%
Splay	22006	21439	2.58%
NavierStokes	32015	30914	3.44%

In addition, we also use V8 Benchmark Suite-Version 7 to test the impact of AntiRead on JIT code, and the results are shown in Table 1. The results show that AntiRead reduces the running speed of JIT code by 3.8% on average.

AntiRead sets the permissions of the code to be unreadable before the process runs. It doesn't use complex mechanisms to adjust code page permissions on-the-fly. Therefore, its overhead on running processes is not so high. It should be noted that the JIT code

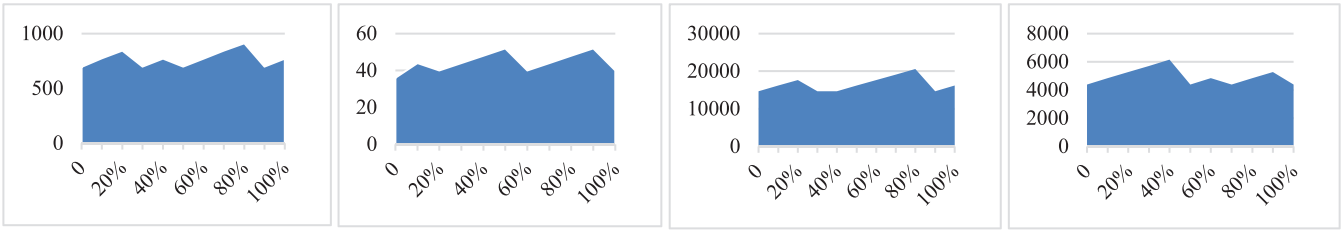

Fig. 15. . The code size during code reading. The horizontal axis represents the reading ratio, and the vertical axis represents the code size (KB). From left to right, the figures are Apache, Sqlite, redis server, and ngnix in sequence.

Table 2
Micro test (ns).

CAddr	CReg	CAR0	CAR^5	CRR^0	CRR^5	ST
2.7	2.8	922.1	2.7	945.5	2.9	516

CAddr: call the code that is never read with call/jmp address; CReg: call the code that is never read with call/jmp *register/pointer; CAR^n: call the code that is read n minutes ago with call/jmp address; CRRn: call the code that n minutes ago with call/jmp *register/pointer; ST: system trap.

can cause two system traps before executing the code. Therefore, AntiRead has a larger impact on it.

To analyze the root cause of the overhead more clearly, we use some micro benchmarks to test AntiRead, as shown in Table 2. The running speed of the code before and after being read will be measured successively. The results show that the operations handling system traps are the main factors introducing overhead. For example, *call *register* triggers a system trap when it jumps to the code that is being read. Therefore, its execution time is longer. In contrast, when it calls the code that has been read 5 minutes ago, there is no system trap needs to be handled by AntiRead. Therefore, its execution time is only 2.9ns.

In addition, the instructions *cpuid, gettsec, invd, xsetbv* and all VMX instructions except *vmfunc* will trigger system traps unconditionally. Their execution frequency will affect the running speed of the process, which is an important factor causing different overhead for different applications.

When code reading occurs, AntiRead allocates readable page(s) in the new space. After the code reading ends, the code page(s) in the new space will be recycled. To verify this conclusion, we use an LKM to read the code and observe the change in code size, as shown in Fig. 15. The results show that the memory overhead introduced by AntiRead does not increase indefinitely. The reason is AntiRead will recycle code pages in the new space with the target process execution. When a system trap occurs, AntiRead checks if the addresses of the code pages that have been read are stored on the stack. If not, the code pages in the new space will be recycled. Meanwhile, the code in the original space will be restored. In contrast, the memory virtualization adopted by AntiRead requires more memory. To cover all physical pages (128GB), two EPTs require about 513MB memory.

Moreover, AntiRead needs to change 650 lines of kernel code and add 400 lines of kernel code, which will not significantly increase the kernel size.

5.3. Comparison with existing methods

We compare the defense effect of existing methods and their performance, as shown in Table 3. The comparison indicates AntiRead can achieve better protection with less overhead. The reason is AntiRead sets the code page as unreadable in advance via the new *buddy system*, which avoids time-consuming tracking of code access and dynamic adjustment of code permissions.

In contrast, the methods relying on source code, such as kRX (Ahmed et al., 2020), are invalid for the shared library code. Except for AntiRead and TASR (Bigelow et al., 2015), all methods directly block code reading activities, or provide a fake code area with them. In the presence of such methods, benign programs cannot get the code with original execution logic. TASR is a randomization method that randomizes the entire code segment when an attacker sends code out. Similarly, AntiRead randomizes the code page being read when the code reading activity occurs. Meanwhile, the code that has been read will be set as non-executable. The legal calls to this page will be transferred to a new space where stores the original code, which can ensure the probed code can still be called normally. Compared with TASR (Bigelow et al., 2015), AntiRead does not need to randomize the entire code segment, nor does it need to restore the complex call relationship between code blocks after randomization. In addition, most methods are invalid for DMA-based code reading probes. With the help of Intel VT-d, AntiRead can hide the target code from DMA activities.

5.4. Limitations

AntiRead still has some limitations. First, it can only protect the indirect pointers stored in code segments and direct pointers stored in GOT and Vtable. It is invalid for the pointers stored in heap, stack and data segments.

Table 3
Comparison with existing methods.

	CA.	RD	CP	AOCR	NS	VL.	NR	O
AntiRead	√	√	*	x	√	√	√	1.8%
XnR[5]	√	x	x	x	√	x	x	2.2%
Heisenbyte[11]	√	x	x	x	√	x	x	16.5%
TASR[14]	√	√	√	√	x	x	√	2.1%
CodeArmor[28]	*	x	√	x	√	√	x	6.9%
Readactor[7]	√	x	√	x	x	x	x	6.4%
HideM[6]	√	x	x	x	√	√	x	2%~6.5%
Near[8]	√	x	x	x	√	√	x	5.7%
KR^X (Pomonis et al., 2017)	√	x	x	x	x	x	x	4.04%

CA: direct code reading; RD: code reading via DMA; CP: code pointer leaks; AOCR: code probing via AOCR (Robert and Skowyra (2017)); NS: effective without source code. VL: valid for library code. O: overhead; NR: no negative impact on legitimate code reading and subsequent execution. *: partially valid.

Second, AntiRead requires the x86 processors equipped with the hardware-assisted virtualization technologies VT-x and VT-d. If there is no such hardware surpport, AntiRead cannot be deployed.

Third, AntiRead cannot defend against AOCR (Robert and Skowyra, 2017). In fact, except for the runtime randomization methods, such as TASR (Bigelow et al., 2015), all methods that restrict code read permissions are invalid for AOCR. AOCR does not need to read any code, which is obviously beyond the protection scope of AntiRead. For runtime randomization methods, such as (Giuffrida et al., 2012; Curtsinger and Berger, 2013; Wang and Wu, 2019; Lu et al., 2016; Wang et al., 2017; Hawkins et al., 2017; Friedman and Musliner, 2015), they are not perfect. They face the problem of selecting randomization points, which directly affects their work efficiency and defense effect. Although frequent randomization can achieve better protection effect, it causes greater overhead. Moreover, the existing randomization methods take the whole code segment or the entire memory object as the target, and they need to solve the complex calling relationship between code objects. More seriously, most fine-grained randomization methods rely on source code, which makes them invalid for closed-source objects.

6. Conclusions

This paper proposes a method AntiRead to prevent adversaries from building gadgets with the code that has been read. Unlike existing methods, it does not completely disable the read access to the code. Instead, it allows any application to read the code, including adversaries and legitimate processes that need to read the code. Once the code is read, it will lose the execution permission in the original space. Then, the code that has been read will be re-randomized in the original space to prevent address leakage. At the same time, the executable code page(s) will be prepared in the new space to ensure that the code that has been read can be called legally. After the code reading is finished, the original code will be mapped back to the original address space. To the best of our knowledge, AntiRead is the first method that can preset all code pages as unreadable, thus achieving better results with less overhead. Experiments and analysis show that AntiRead can prevent the code that has been read from being used as gadgets without affecting other applications to read the code legally. Furthermore, AntiRead introduces 1.8% overhead to CPU.

Declaration of Competing Interest

We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work. And there is no professional or other personal interest of any nature or kind in any product, service and/or company that could be construed as influencing the position presented in, or the review of, the manuscript entitled, “What you can read is what you can’t execute”.

CRediT authorship contribution statement

YongGang Li: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing, Project administration. **JiaZhen Cai:** Formal analysis, Investigation. **Yu Bao:** Data curation, Writing – review & editing. **Yeh-Ching Chung:** Writing – review & editing, Supervision, Funding acquisition.

Data availability

No data was used for the research described in the article.

Acknoelegment

This work is supported by Jiangsu Province Double-innovation Doctor Project, No. 140923050.

References

Jang, D., 2022. Badaslr: exceptional cases of ASLR aiding exploitation. Comput. Secur. 112. doi:10.1016/j.cose. 2021.102510, Jan..

Lu, K., Xu, M., Song, C., Kim, T., Lee, W., 2021. Stopping memory disclosures via diversification and replicated execution. IEEE Trans. Dependable Secure Comput 18 (1), 160–173. doi:10.1109/TDSC.2018.2878234, Jan..

Zhang, M., Polychronakis, M., Sekar, R., 2017. Protecting COTS binaries from disclosure-guided code reuse attacks. In: Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC’ 17), New York, NY, USA, pp. 128–140.

Hu, H., Shinde, S., Adrian, S., Chua, Z.L., et al., 2016. Data-oriented programming: on the expressiveness of non-control data attacks. In: Proceedings of the IEEE Symposium on Security and Privacy (SP), pp. 969–986.

Backes, M., Holz, T., Kollenda, B., et al., 2014. You can run but you can’t read: preventing disclosure exploits in executable code. In: Proceedings of the ACM SIGSAC Conference Computer and Communications Security (CCS ’14), New York, NY, USA, pp. 1342–1353.

Gionta, J., Enck, W., Ning, P., 2015. HideM: protecting the contents of userspace memory in the face of disclosure vulnerabilities. In: Proceeding of the 5th ACM Conference Data and Application Security and Privacy (CODASPY ’15), New York, NY, USA, pp. 325–336.

Crane, S., et al., 2015. Readactor: practical code randomization resilient to memory disclosure. In: Proceedings of the IEEE Symposium Security and Privacy, pp. 763–780. doi:10.1109/SP.2015.52.

Werner, J., Baltas, G., Dallara, R., et al., 2016. No-execute-after-read: preventing code disclosure in commodity software. In: Proceedings of the 11th ACM on Asia Conference Computer and Communications Security (ASIA CCS ’16), New York, NY, USA, pp. 35–46.

Kjell, B., Lucas, D., Christopher, L., et al., 2016. Leakage-resilient layout randomization for mobile devices. In: Proceeding of the NDSS, 16, pp. 21–24.

Pomonis, M., Petsios, T., Keromytis, A.D., Polychronakis, M., Kemerlis, V.P., 2017. KR^X: comprehensive kernel protection against just-in-time code reuse. In: Proceedings of the 12th European Conf. Computer Systems (EuroSys ’17), New York, NY, USA, pp. 420–436.

Tang, A., Sethumadhavan, S., Stolfo, S., 2015. Heisenbyte: thwarting memory disclosure attacks using destructive code reads. In: Proceedings of the 22nd ACM SIGSAC Conf. Computer and Communications Security (CCS ’15), New York, NY, USA, pp. 256–267.

Crane, S., Homescu, A., Larsen, P., 2016. Code randomization: haven’t we solved this problem yet? In: Proceeding of the IEEE Cybersecurity Development (SecDev), pp. 124–129.

Li, Y.G., Guoyuan, L., et al., 2023. MagBox: Keep the risk functions running safely in a magic box. Future Gener. Comput. Syst. 140, 282–298.

Bigelow, D., Hobson, T., Rudd, R., et al., 2015. Timely rerandomization for mitigating memory disclosures. In: Proceedings of the ACM conf. Computer and communications security (CCS ’15, pp. 268–279.

Conti, M., Crane, S., Frassetto, T., et al., 2016. Selfrando: Securing the tor browser against de-anonymization exploits. In: Proceedings of the PETS.

Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z., 2012. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of the ACM Conference Computer and Communications security (CCS ’12), New York, NY, USA, pp. 157–168.

Robert, R., Skowyra, R., et al., 2017. Address oblivious code reuse: on the effectiveness of leakage resilient diversity. In: Proceedings of the NDSS.

Hiser, J., Nguyen-Tuong, A., et al., 2012. ILR: Where’d My Gadgets Go? In: Proceedings of the IEEE Symposium Security and Privacy, pp. 571–585.

Chen, Y., et al., 2017. NORAX: enabling execute-only memory for COTS binaries on AArch64. In: Proceedings of the IEEE Symposium Security and Privacy (SP, pp. 304–319.

Ahmed, S., Xiao, Y., Snow, K.Z., et al., 2020. Methodologies for quantifying (Re-)randomization security and timing under JIT-ROP. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 1803–1820.

Chen, Y., Wang, Z., Whalley, D., Lu, L., 2016. Remix: On-demand Live Randomization. In: Proceedings of the ACM Conference on Data and Application Security and Privacy, pp. 50–61.

Zhang, L., Choffnes, D., Levin, D., Dumitra, T., et al., 2014. Analysis of SSL certificate reissues and revocations in the wake of heartbleed. In: Proceedings of the Internet Measurement Conference, pp. 489–502.

Liljestrand, H., et al., 2019. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In: Proceedings of the 28th USENIX Security Symposium (USENIX Security 19.

Li, Y.G., Chung, Y.C., et al., 2022. KPointer: Keep the code pointers on the stack point to the right code. Comput. Secur., 102781.

Li, Y.G., Chung, Y.C., Xing, J., et al., 2022. MProbe: Make the code probing meaningless. In: Proceedings of the 38th Annual Computer Security Applications Conference, pp. 214–226.

Nikolaev, R., Nadeem, H., et al., 2022. Adelie: continuous address space layout re-randomization for linux drivers. In: Proceedings of the ACM International

Conference on Architectural Support for Programming Languages and Operating Systems, pp. 483–498.
Williams-King, D., Gobieski, G., Williams-King, K., et al., 2016. Shuffler: fast and deployable continuous code {re-randomization}. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 367–382.
Chen, X., Bos, H., Giuffrida, C., 2017. CodeArmor: virtualizing the code space to counter disclosure attacks. In: Proceedings of the IEEE European Symposium on Security and Privacy, pp. 514–529.
Davi, L., Gens, D., Liebchen, C., et al., 2017. PT-rand: practical mitigation of data-only attacks against page tables. In: Proceedings of the NDSS.
Lu, K., Song, C., Lee, B., et al., 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, pp. 280–291.
Rajasekaran, P., Crane, S., Gens, D., et al., 2020. CoDaRR: Continuous data space randomization against data-only attacks. In: Proceedings of the15th ACM Asia Conference on Computer and Communications Security, pp. 494–505.
Backes, M., Nürnberger, S., 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In: Proceedings of theUSENIX SECURITY, pp. 433–447.
Sun, M., Lui, J.C., et al., 2016. Blender: Self-randomizing address space layout for android apps. In: Proceedings of the RAID, pp. 457–480.
Giuffrida, C., Kuijsten, A., Tanenbaum, A.S., 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In: Proceedings of the USENIX SEC, pp. 475–490.
Curtsinger, C., Berger, E.D., 2013. Stabilizer: Statistically sound performance evaluation. In: Proceedings of the ACM SIGARCH Computer Architecture News, 41, pp. 219–228.
Wang, Z., Wu, C., et al., 2019. {SafeHidden}: An Efficient and Secure Information Hiding Technique Using Re-randomization. In: Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), pp. 1239–1256.
Lu, K., et al., 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In: Proceedings of the NDSS.
Wang, Z., Wu, C., Li, J., et al., 2017. Reranz: A light-weight virtual machine to mitigate memory disclosure attacks. In: Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 143–156.
Hawkins, W., Nguyen-Tuong, A., Hiser, J.D., et al., 2017. Mixr: Flexible runtime rerandomization for binaries. In: Proceedings of the Workshop on Moving Target Defense, pp. 27–37.
Friedman, S., Musliner, D., et al., 2015. Chronomorphic programs: runtime diversity prevents exploits and reconnaissance. In: Proceedings of the International Journal on Advances in Security, pp. 120–192.
Schloegel, M., Blazytko, T., Basler, J., et al., 2021. Towards automating code-reuse attacks using synthesized gadget chains. In: Proceedings of the European Symposium on Research in Computer Security. Springer, Cham, pp. 218–239.
J. Salwan, 2023 “ROPgadget–Gadgets Finder and Auto-Roper,” http://shell-storm.org/project/ROPgadget.
Yun, J., Park, K.W., Koo, D., et al., 2020. Lightweight and seamless memory randomization for mission-critical services in a cloud platform. Energies 13 (6), 1332.

Yong-Gang Li, received the PhD degree from the University of Science and Technology of China in 2019. He was a postdoctoral fellow in the Chinese University of Hong Kong, Shenzhen. Now, he is an associate professor with the School of Computer Science and Technology in the China University of Mining and Technology (CUMT). His research interests include computer architecture, virtualization principle, cloud computing, and system security.

JiaZhen Cai, is an ungraduated student at the School of Computer Science and Technology in CUMT. His research interests include system optimization and security.

Yu Bao, received the PhD degree from Tongji University in 2011. Now, he is an associate professor with the School of Computer Science and Technology in CUMT. His research is information security in IoT.

Yeh-Ching Chung, received Ph.D. degrees in Computer and Information Science from Syracuse University in 1992. Now, he is a Professor of the Chinese University of Hong Kong, Shenzhen. His research interests include parallel and distributed processing and system software.