MProbe: Make the code probing meaningless

Yonggang Li
The School of Computer Science and Technology, the China University of Mining and Technology
liyg@cumt.edu.cn

Ye-Ching Chung
The School of Data Science, CUHK(SZ)
ychung@cuhk.edu.cn

Jinbiao Xing
The School of Computer Science and Technology, the China University of Mining and Technology
jbxing@mail.ustc.edu.cn

Yu Bao
The School of Computer Science and Technology, the China University of Mining and Technology
baoyu@cumt.edu.cn

Guoyuan Lin
The School of Computer Science and Technology, the China University of Mining and Technology
lingy@cumt.edu.cn

ABSTRACT

Modern security methods use address space layout randomization (ASLR) to defend against code reuse attacks (CRAs). However, code probing can still obtain the content and address of the code through code probing. Code probing invalidates the widely used ASLR methods, causing researchers to lose confidence in them. On the contrary, we believe the ASLR is still effective, if it has anti-probing capability. To enhance the anti-probing capability of ASLR and defense CRAs, this paper proposes an anti-probing method MProbe. First, it detects the code probing activities of attackers, including address probing and content probing. Next, the execution permission of the probed code will be de-enabled in the original address space. At the same time, the equivalent code block in a random address space will replace the probed code. Finally, new security strategies are used to prevent the probed code blocks from being used as gadgets. Experiments and analysis show that MProbe has a good defense effect against CRAs based on code probing, and only introduces less than 3% performance overhead to the operating system (OS).

CCS CONCEPTS

• **Security and privacy** → Systems security; Operating systems security; Virtualization and security.

KEYWORDS

Integrity, System architectures, Security and Protection

ACM Reference Format:
Yonggang Li, Ye-Ching Chung, Jinbiao Xing, Yu Bao, and Guoyuan Lin. 2022. MProbe: Make the code probing meaningless. In *Annual Computer Security Applications Conference (ACSAC '22), December 05–09, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3564625.3567967

1 INTRODUCTION

CRAs[1,2] bring challenges to the OS security. They do not rely on the injected code, causing the DEP to fail[3]. The difference between CRA variants and legal code is gradually shrinking. Therefore, the signature-based methods[4] are losing their effects. Although the security methods based on the control flow feature[5] can identify CRAs, they not only face the state explosion of control flow paths, but also introduce huge performance overhead. The boundary detection methods[6] cannot provide precise boundaries for indirect control transfer (ICT) instructions whose control data is dynamic, which reduces their protection effects.

ASLR[7] makes it possible to have a security method considering both execution efficiency and defense effects. It randomizes memory layout, so that attackers cannot obtain the address of the gadgets[8]. ASLR only hides the address of the code or control data, and does not intervene in the execution of execution entities. So, it has high execution efficiency. In ASLR environment, attackers cannot build gadget chains due to lack of the knowledge of code content and code addresses.

Unfortunately, the probing attack[9] can still get code content and code addresses. Although the periodic or real-time ASLR[10] has a certain degree of anti-probing ability, it makes ASLR lose performance advantage. Facing the challenge of probing attacks, ASLR seems to have gradually lost its defense effect. However, we believe ASLR is still an excellent security method, if it has anti-probing capability. Around this point of view, this paper proposes an anti-probing method MProbe.

The goal of a probing activity is to get the code content or code addresses required by CRAs. Then, the probed code blocks conforming specific forms (such as *pop rax; jmp *rax*) will be connected together to form a gadget chain. In real attack scenarios, both the probing activities and the control flow along the gadget chain have abnormal behavior characteristics. These characteristics are key to detecting and defensing CRAs.

There are two probing targets for CRAs, code address and code content. Among the currently known probing technologies, side channel[37] and data-leak[38] can directly get the code address. Arbitrary read[39] and arbitrary jump[16] can obtain the specific code forms. Additionally, arbitrary write and process cloning[15] can be used as aids to probe code content and memory layout.

Due to ASLR, code probing has become a prerequisite to implement CRAs. For the memory space with fine-grained ASLR, it is impossible to get all gadgets through a single probing in the random space. For example, BROP[16] needs to repeatedly modify the return addresses according to the crash information to find available gadgets. The primary task of MProbe is to perceive the probing behavior of attackers and get the probed code.

No matter what probing it is, there are differences between malicious code probing and normal memory access. MProbe can perceive the attacker's probing activities through these differences.

The probed code may be used as a gadget by the attacker. In a real attack scenario, the gadget may be a node in the entire gadget chain, which is used to connect its adjacent nodes; it may also be a dispatcher gadget[40] , which is used to transfer control flow to each node. The second task of MProbe is to prevent the probed code from being used as a gadget or a dispatcher gadget.

ICT instructions are key points to connect gadgets. Both the operand of the ICT instruction, such as *call *rsi*, and the return address of the instruction ret is an absolute address. Therefore, whether it is a gadget or a dispatcher gadget, it will be called in the original address space. So, MProbe can prevent them from being maliciously used by striping the execution permission of the probed code in the original address space.

In fact, the probed code may still be called legally by the process. However, it cannot be called in the original space due to the inexecutable permission, even if the call is legal. The third task of MProbe is to ensure the probed code can be called legally.

Since the probed code cannot be called in the original address space, MProbe designs a new address space for it. When the legal control flow is transferred to the new space, it can flow smoothly. When the control flow is transferred to the probed code blocks through ICT instructions or ret, its legitimacy will be judged according to a series of security strategies. In summary, MProbe can defend against CRAs relying on code probing, and its contributions are as follows:

A. Propose a probing perception mechanism. This mechanism is on the basis of fine-grained ASLR. When an attacker attempts to reveal memory layout or collects available gadgets, it can perceive the probing activity.
B. Propose a protection mechanism to prevent the probed code from being used as gadgets. This mechanism turns off the execution permissions of the probed code in the original address space. At the same time, the equivalent code block in the new address space will replace the probed code to ensure the probed code can be called legally.
C. Implement the MProbe prototype in Linux. MProbe can detect code probing in real time. It can also defense CRAs through new security strategies, and only introduces less than 3% performance overhead to the OS.

2 RELATED WORKS

Researchers have carried out many researches on CFI protection, mainly including anti-probing and control flow path protection. In this section, we describe the two methods separately.

2.1 Anti-probing methods

ASLR changes the memory layout of the target to be protected. CCFIR[11] is a coarse-grained randomization method for binary executable, and it can be bypassed if there exists memory leakage[12]. Marlin[13] is a fine-grained ASLR method. It decomposes the binary file of the application into multiple parts with functions as code blocks, and then confuses all parts. The randomization granularity of ILR[14] is smaller, and it can randomize every instruction in the program. The ultimate purpose of ASLR is to make the gadget address obtained through static analysis unable to be used by attackers during the execution of the entity. However, the attacks, such as Clone-ROP[15] can obtain the address information by cloning the address space of the parent process. In addition, attackers also use methods such as arbitrary jump[16], arbitrary read[39], and crash-less counterparts[18] to probe code information. These methods can also bypass ASLR.

To solve the problem of ASLR being bypassed, researchers hide or encrypt the control data. BarRA[20] destroys the original return address and allocates a new one after detecting the return address leakage. However, it is only valid for the return address and invalid for other control data, such as function pointers. To prevent more control data from being probed, memory hiding[21, 22] are widely used. CPI[23] and ASLR-Guard[24] hide the information related to function pointers. Isomeron[25] and Oxymoron[26] hide the run-time lookup tables related to code randomization. Unfortunately, these methods can be easily bypassed in the face of memory leak. For example, AOCR[17] and CROP[19] can still get available gadgets in a hidden space.

2.2 Control flow path protection

The hijacked control flow inevitably changes the original control flow paths. Researchers use the path restriction[27,28,29,30,31] to prevent control flow from jumping out of the specified range. μCFI[27] is a fine-grained CFI method. It identifies sensitive instructions and checks the program to record the necessary execution context. It monitors the program in different processes, and interprets sensitive instructions in the recorded execution context. However, this method is invalid for the loaded library code (without source code). πCFI[29] is also a fine-grained method. It can dynamically generate control flow graphs, avoiding the problem of state explosion. However, it will derive a high-risk attack surface because of opening the write permission of code, and pose a huge threat to system security. Moreover, πCFI also requires source code, which limits its protection scope.

There also exist many coarse-grained CFI methods including KCoFI[32], binCFI[33], and O-CFI[34], etc. Their control flow graphs are easier to build, even without access to source code. But on the down side, the coarse-grained control flow graphs are too permissive so that it is still possible to mount attacks in general.

3 THE OVERALL DESIGN OF MPROBE

3.1 Assumptions and threat models

First, we assume the fine-grained ASLR with basic code block containing only one exit is in use, and attackers cannot infer all gadgets' locations from a leaked code address. In practice, fine-grained randomization has been matured, and their targets cover pages, functions, basic blocks and instructions. Second, we assume attackers can probe memory repeatedly. For some probing technologies, they will trigger exceptions when probing the code space. For example, the arbitrary jump[16] may be transferred to unmapped areas, which causes segment faults. However, the OS allows applications to handle exceptions by themselves to avoid process crash. So, attackers can repeatedly probe memory without worrying about probing activities being suspended. Third, we assume attackers can hijack the control flow by modifying return addresses or function pointers through vulnerabilities.

The threat model used in this paper includes 4 attack vectors:

Vector 1: Arbitrary Read[39]. Attackers can exploit vulnerabilities (such as HeartBleed) read the memory in the code segment. Arbitrary read can be attacked in two ways. One is to start from the data area and gradually move closer to the code segment until the code is read. The other is to use relative addresses in the code segment to read more code pages recursively.

Vector 2: Arbitrary Jump[16]. Attackers can redirect the control flow to any position of the memory by tampering with the control data, and find the available gadgets by analyzing the crash information caused by the redirected flowing flow.

Vector 3: Side-channel Probing[37]. Attackers can get the 12th to 47th bits of the code address by analyzing the hit information of the translation lookaside buffer (TLB).

Vector 4: Data-Leak[38]. Attackers can exploit DOP[38] to read the relative offset in PLT (Procedure Linkage Table), and then they can get the random address of GOT (Global Offset Table), where stores the address of library functions.

In addition to these 4 attack vectors, attackers can also use the allocation oracles[48], process cloning[15] and arbitrary write[41] (such as stack overflow) as auxiliary means for code probing, which can help attackers reduce the difficulty of probing.

3.2 Overall architecture of MProbe

The main idea of MProbe is to make the probed code lose its execution permission in the original address space. As a result, although attackers can obtain the code information, they cannot execute it ICT instructions and ret. To achieve this goal, MProbe faces three challenges: perceive the code probing activities, prevent the probed code snippets from being used as gadgets, and ensure the probed code to be called legally.

The overall architecture of MProbe is shown as Figure 1. MProbe is composed of 4 components: probe-wall, permission manager, security engine and CF transfer. The probe-wall can detect the probing activities. If the code is being probed, the permission manager disables the execution permission of the probed code. At the same time, the probed code (rather than all the code) will be migrated to a random address space, where the code is executable. Then, the security engine will judge the legitimacy of the current control flow. To handle the legal call to the probed code, CF transfer redirects the control flow to the random space.

The operations of MProbe on the code include probing detection, code rewriting, legitimacy judgment and control flow transfer. All these need the capabilities that monitor and control the activities of execution entities (especially the ones without source code, such

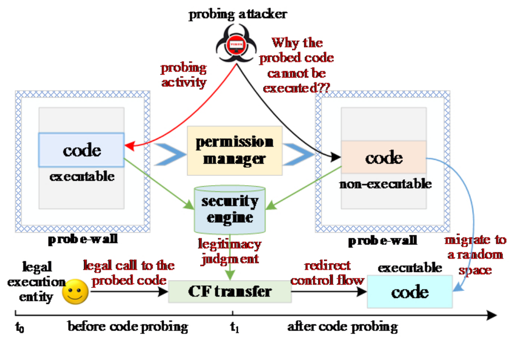

Figure 1: The overall architecture of MProbe

as the loaded library code) in real time. The memory virtualization based on EPT (Extended Page Tables) and the event mechanism based on VMX (Virtual Machine Extension) will be adopted to support MProbe's implementation.

The Intel VMX non-root and VMX root will be used to divide the OS into two modes: guest and host. Under normal cases, the OS runs in guest. When a specific event (such as code reading) occurs, the OS falls into the host, which is called system trap in this paper. In host mode, MProbe can take over the control flow of the entire OS. Besides, we can use EPT to manage all memory in the OS, including permission (read, write, and execute) management and space (physical space and virtual space) management. Any memory operation that violates the management strategies will trigger a system trap, which will be captured by MProbe. Moreover, some specific events (such as *int3*, *vmcall*, and *mov to cr3*) can be monitored and controlled by manipulating the fields in the VMCS (virtual machine control structure).

4 PERCEIVE THE PROBING ATTACKS

The probe-wall is used to perceive the code probing of attackers. The attack vectors in 3.1 exploit different probing techniques, which pose challenges to the probe-wall.

Perceive Vector 1. To perceive the Vector 1, MProbe directly disables the read permissions of the application code and library code. Therefore, the attack will cause a permission exception when reading code, which will be perceived by MProbe.

Unfortunately, we cannot directly control the read permissions of memory. Although the existed methods can indirectly control the read permission by setting the president bit (*p* bit) of the page tables, they seriously affect the OS performance. EPT provides the possibility to control the read permissions directly. We can open and close the read permission of physical pages through the r bit in the last-level page table of EPT. However, there are still many problems in operating the physical pages of the code.

First, the OS does not load all code into the memory at one time. It loads the code into the memory when the code is called for the first time. So, we don't know the corresponding physical addresses of the code before it is executed, and we can't set it to be unreadable in advance. Second, all the physical memory is shared. A physical page will be used to store data or code at different periods. For example, after a process dies, its code pages may be used to store

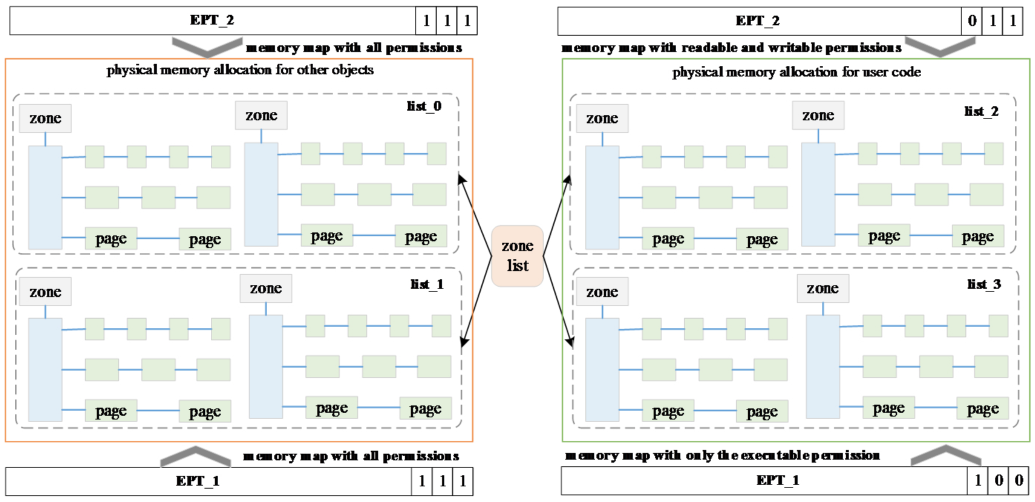
EPT_2
1 1 1
memory map with all permissions
physical memory allocation for other objects
zone
page
list_0
list_1
zone list
EPT_2
0 1 1
memory map with readable and writable permissions
physical memory allocation for user code
list_2
list_3
memory map with all permissions
EPT_1
1 1 1
memory map with only the executable permission
EPT_1
1 0 0

Figure 2: The overall design of user code memory allocation

data of other processes. If we set the code page to be unreadable and cannot change it to be readable when it is recycled, then other execution entities will inevitably cause a memory exception when reading the data in the memory.

The most direct way to solve the above problems is to track every physical page allocation. By tracking all page faults in the OS, we can identify the code page one by one. However, the frequency of page faults in the OS is very high, and capturing all page faults is expensive. In our test, we captured all page allocations by hooking the function *do_page_fault*. We found this method caused a 50%~350% reduction for the running speed of processes (such as Nbench). Even if the *PFEC_MASK* and *PFEC_MATCH* fields in VMCS can be used to only capture the code page allocations, the running speed of the process slows down by 30% to 170%. Such performance loss is unacceptable.

In Linux, memory allocation is done by the buddy system. Physical pages are organized and managed by *zone list*, *zone* and *page*. In the native OS of the NUMA architecture with two CPUs, the buddy system uses two *zone lists* for each node to manage all *zones*. The first *zone list* is used to manage the *zones* connected to the current CPU, and the second is used to manage the *zones* in all CPUs. When there is a page fault in code segment, the buddy system will select a specific number of physical pages from a *zone* in the *zone list* (preferably *list_0*).

We combine EPT to modify the buddy system to reduce the performance loss caused by perceiving code reading. This is shown in Figure 2. It doubles the number of the original *zone lists*. In the new buddy system, *list_0* and *list_1* are used to allocate memory for all Linux objects except user code, while *list_2* and *list_3* are only used to allocate physical pages for user code. *list_0* and *list_2* divide the memory directly connected to the current CPU into two parts evenly. *list_1* and *list_3* divide all the memory into two parts evenly.

We set the memory in list_0 and list_1 to be readable, writable and executable through EPT. Therefore, accessing the memory will not be affected in any way. To capture the attacker's code reading, we must set the code to be unreadable before the probing attack occurs. At the same time, to ensure the OS can load the code into the memory, we also need to set the physical page to be writable when the code is loaded. However, setting a physical page as writable and unreadable at the same time in EPT will cause EPT misconfiguration. To solve this problem, we use *EPT_1* to set the physical memory in *list_2* and *list_3* to be executable only, and use *EPT_2* to set the physical memory in *list_2* and *list_3* to be readable, writable but non-executable.

The process won't read its own code. However, there are mixed pages containing both code and data. For such pages, when deploying ASLR, the base address is adjusted so that data and code are stored in different pages. Therefore, data and code can be set to different permissions.

When the OS loads the user code, it calls the function *filemap_fault*. Therefore, we perform EPT switching by adding a hook in *filemap_fault*. If the *filemap_fault* is triggered by a code page fault, we will switch EPT to *EPT_2* by executing the instruction *vmfunc*, and switch EPT back to *EPT_1* when *filemap_fault* returns. When the OS attempts to load user code, a physical page will be selected from *list_2* or *list_3* that has been set to be readable and writable but non-executable. When the code has been loaded into memory, the code page will become executable but unreadable and unwritable. Therefore, when a code reading occurs, a system trap will be triggered, which will be captured by MProbe. In a word, MProbe can set the code to be unreadable before it is executed without tracking all code pages' allocation like NEAR[49], or capturing every code page access like XnR[50].

In summary the buddy system is modified to create a memory pool, which is the source of code page allocation. Pages in this pool are pre-set as unreadable. So, we can prevent code reading probes without tracking page allocation and reading activities. System traps are triggered whenever abnormal activities occur. Then we can analyze the current state and history activities of the process.

Perceive Vector 2. Vector 2 searches for available gadgets by analyzing the crash information caused by arbitrary jumps, which usually requires the assistance of arbitrary writes. Due to ASLR, arbitrary jumps are likely to jump into unmapped space or into illegal binary code. The former triggers the signal *SIGSEGV* and the latter triggers the signal *SIGILL*. In user space, the available address space is 128TB. The code segment is only a small part in it, and the vast majority of the areas are unmapped. For example, a 128MB code segment occupies only one millionth of the entire space. Therefore, if the address knowledge is unknown due to the fine-grained ASLR, an arbitrary jump has a high probability of transferring the control flow to an unmapped area. Even if an attacker is able to transfer the control flow to the mapped code segment, he cannot precisely transfer it to the available gadgets. According to our observations, the arbitrary jumps within the code segment have a high probability of triggering illegal instructions. We simulate BROP by arbitrarily tampering with the last 12 bits of the return addresses. The results show that the probability of triggering illegal instructions exceeds 99% for three consecutive control flow transfers.

If the signal *SIGSEGV* or *SIGILL* appears, the current process will be killed. However, the OS allows applications to handle such signals themselves to avoid process crashes. As a result, attackers can repeatedly probe memory without process crashes. Moreover, process cloning can also help an attacker to repeatedly probe memory, which can avoid the parent process's crash.

To perceive Vector 2, MProbe modifies the system call *signal*, which is used by the application to handle signals itself. If the signal handled by the system call *signal* is *SIGSEGV* or *SIGILL*, the instruction that triggers the signal will be judged as a probing instruction. Even a child process obtained by process cloning can be captured when it triggers *SIGSEGV* or *SIGILL*. After that, MProbe locates the same instruction in its parent process (if it exists) according to the probing instruction of the child process.

If the application does not define a signal handler, the OS will call *do_coredump* by default to handle the signals *SIGSEGV* and *SIGILL*. After that, the process will be killed. However, an attacker can repeatedly probe memory by restarting the process again and again until getting available gadgets. To perceive this attack, we add a hook at the kernel function *do_coredump* to get its parameters *signr* and *regs*, which store the signal type and the code address that triggers the signal, respectively. If the signal being handled is *SIGSEGV* or *SIGILL*, we compute a hash value for the code block (*CB_n*) triggered the signal. At the same time, we also record 128 bytes of code (*C_n*) at a random location (*L_n*) in the process code segment. When *do_coredump* processes the signal *SIGSEGV* or *SIGILL* again, we first calculate the hash value of the code block that triggers the signal, and compare it with the recorded hash values. If there is a same hash value *CB_n*, we need to judge whether the current process is a restarted process by comparing *C_n* with the current process's code at *L_n*. If they are same, we record the code block and modify the ELF file of the process to mark it with *int3*. When the process is restarted again, calling the marked code block triggers a system trap. After that, MProbe will use security policies to check its legitimacy, which will be introduced in Chapter 5.

Perceive Vector 3. Vector 3 exploits side-channel to crack the 12th to 47th bits of the random address. In the process of converting a virtual address to a physical address, the 12th to 20th, 21st to 29th, 30th to 38th, and 39th to 47th bits of the virtual address respectively indicate the index values of the page tables at 4 levels. Each entry in the 1st to 4th level page tables can index 4KB, 2MB, 1GB, and 512GB memory, respectively. To obtain the last 3 bits of the index value of the page table at each level, Vector 3 needs to access the memory which is separated from the virtual address by *n*4KB, n*2MB, n*1GB*, and *n*512GB* respectively (*n<8*). For example, to get the index value of the virtual address *V* in the third-level page table, Vector 3 needs to execute *V+n*1GB* (*n=1, 2, 3...*) in sequence until the current cache line is filled. In practice, the size of the code segment rarely exceeds 1GB, and even less than 512GB. So, to execute the code at *V+n*xGB*, the attacker needs to allocate a new area at *V+n*xGB*, where the code is executable.

If the space at *V+n*xGB* (*V∈code segment, n<8, x=1 or 512*) is unmapped, MProbe uses the kernel function *do_mmap* to map the space and sets its corresponding page tables to be unreadable through EPT. It should be noted that we just allocated some page tables for *V+n*xGB*, without allocating other physical memory for it. If the space at *V+n*xGB* has been mapped, MProbe also sets the page table corresponding to the space to be unreadable. When there is a memory access to this area, MProbe enables the read permission of the page tables and records the accessed memory address. Note that the cloned process also has the same permission configurations. MProbe can perceive the memory probing when Vector 3 cracking the virtual address, regardless of whether the area at *V+n*xGB* has been mapped or not.

Perceive Vector 4. Vector 4 exploits data leak to read the GOT address stored in PLT. Then, it can read the addresses of library functions stored in GOT with the same way. MProbe has set the code segment including PLT to be unreadable (described in the section Perceive Vector 1). So, attackers cannot read the GOT address (relative offset) stored in PLT.

To prevent attackers from directly reading the library function addresses in GOT, MProbe sets GOT to be unwritable when the process starts, which is shown in Figure 3. Therefore, each library function address can be captured by MProbe when it is written to the GOT. Then, the library function address (*real address*) will be written to a non-readable code snippet (*secret code*). After that, each library function address written to the GOT is tracked and recorded. The real library function address will be stored in an unreadable area (*secret code*) to prevent the leakage of the library function addresses. Finally, the code snippet's starting address (*L0*) will be filled in GOT. As a result, Vector 4 cannot get the address of the library function by reading GOT.

In addition to the above 4 attack Vectors, allocation oracles, process cloning and arbitrary writing can also be used to probe the memory. But none of them can directly get available gadgets or accurate code address. They need to be combined with other probing

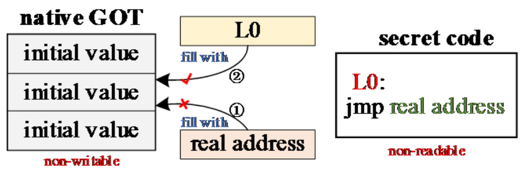

Figure 3: Hide the function address in GOT

technologies to complete the final probing purpose. For example, after the process cloning gets a child process, the attacker still needs to read code to obtain the code information of the parent process. Therefore, MProbe does not treat the three probing technologies as the perceiving targets, but it can still perceive the memory probing activities when the three probing technologies are combined with others.

5 PREVENT THE PROBED CODE IS USED AS A GADGET

In the real attack scenario, when the code probing is perceived, some code snippets may have been probed. For example, arbitrary jump may have been executed several times before it is perceived by MProbe. In special cases, some code snippets can even be used as gadgets directly without probing. For example, an attacker can modify the function pointer by exploiting a heap overflow, and then get an available gadget *call *rax*. Although some gadgets can be used without probing, they cannot constitute a complete gadget chain. Some of them will be used as a probing tool, and others will be used as a node in the gadget chain after probing. In a word, whether it is the code that has been probed or can be used as a gadget without probing, it is dangerous. Therefore, we must stop the probed code from being used as gadgets or probing tools. Whether it is used to probe code or connect other gadgets, its malicious purpose can be detected by MProbe.

When the Vector 1 is perceived, MProbe directly prevent the current code reading. Generally speaking, the process does not read its own code or library code. Therefore, the prohibition of code reading will not affect the execution of the process.

When the Vector 2 is perceived, the jump activity may be attacker's probing activity, or it may only be a mistaken operation of the current process. In fact, it has a certain risk whether it's a mistaken operation or a malicious operation. Even if the current jump is malicious, it may not be the root cause of this abnormal execution. The reason is that other malicious jump instructions may have been executed multi times before the current jump instruction is executed. For example, BROP can find the code snippet *ret* by modifying return addresses, and several *ret* may have been located before the signal *SIGSEGV* or *SIGILL* is triggered. Therefore, we cannot directly prohibit the current instructions from being executed like processing Vector 1. Because this will not only affect the probed code being legally called, but may also miss the root cause of abnormal execution.

When Vector 3 is perceived, it means that the attacker may have found a code snippet that conforms to a specific gadget form. Available gadgets are stored in the memory being cracked. At this point, the attacker is trying to crack the 12th~14th, 21st~23rd, 30th~32nd, and 39th~41st bits of the virtual address. In addition to these bits, the others in the 12th~47th bits have been cracked. That is, for the memory being probed, most of its address has been leaked. Attackers can further crack the remaining bits exploiting process cloning or malloc oracles, and the difficulty of cracking decreases exponentially. Like Vector 2, we also cannot directly prohibit the probed code from being called.

When Vector 4 is perceived, the attacker is either reading the PLT or the *secret code* (see Figure 3). In a legitimate scenario, neither the PLT nor the *secret code* will be read by process. Therefore, MProbe can directly prohibit the current read activity without affecting the normal execution of the process.

To sum up, for Vector 1 and Vector 4, MProbe prevents attackers from deploying CRAs by prohibiting the current probing activity. Because Vector 1 and Vector 4 will be perceived by MProbe at the beginning of code probing. At this point, attackers have neither available gadgets nor leaked address. For Vector 2 and Vector 3, prohibiting their probing activity does not provide sufficient security for the process. Because, when Vector 2 and Vector 3 are perceived by MProbe, attackers have found available gadgets, or got most bits of the address. Therefore, MProbe needs to prevent them from being used as gadgets.

The gadget contains a control flow transfer instruction, and the transfer target is stored in writable memory. These instructions (called ICT instructions in this paper) include *call *register*, *call *(register)*, *call *value(register)*, *call *(register1, register2, value)*, *call *pointer*, *jmp *register*, *jmp *(register)*, *jmp *address (, register, value)*, *ret*, *retn value*, and *retf value*. In Vector 2 and Vector 3, the code block containing an ICT instruction is what MProbe needs to monitor and protect. Such a code block starts with the next instruction of a control flow transfer instruction (such as *call address*) and ends with an ICT instruction.

When Vector 2 is perceived because of *system traps*, MProbe analyzes the binary code to find the code block caused the execution exception. This code block may just be a node in the entire probing chain. Another word, before the current code block is executed, the attacker may have probed the code space multiple times and got several available gadgets. Only the current code block triggering an exception is captured by MProbe. Therefore, MProbe needs to backtrack the control flow transfer to detect whether the attacker has found available gadgets.

MProbe uses LBR registers to look up the last 16 pairs of jump instructions to find the 16 most recently executed code blocks. By setting the *MSR_LBR_SELECT* field in LBR registers, MProbe only captures ICT instructions in user code (ring>0). In addition, *call address* will also be captured. In theory, attackers may have executed more than 16 code blocks containing ICT instructions or *call address* before being perceived. In such execution scenario, only analyzing the LBR register cannot get all the probed code blocks. Although Intel BTS can solve this problem, it significantly slows down the execution speed of the process. In contrast, the overhead introduced by LBR is almost negligible. Fortunately, we found that, under the protection of fine-grained ASLR and MProbe, Vector 2 cannot execute 16 illegal code blocks consecutively without triggering a *system trap*. Under the ASLR and MProbe, we use *redis, tar, httpd*, and *Nginx* as attack objects respectively, and simulate BROP attack by manually tampering with the return addresses. When return addresses are arbitrarily rewritten, the probability of triggering a *system trap* for the first execution of the instruction *ret* exceeds 99%. When only the last 12 bits of return addresses are tampered with arbitrarily, the instruction *ret* calls an average of 1.27 code blocks when triggering a *system trap*. In our 4000 tests, BROP can only execute at most 5 code blocks without triggering a *system trap*. In most cases, it will be captured by MProbe after probing only one code block. Therefore, we conclude that 16 pairs of LBR registers are enough for MProbe.

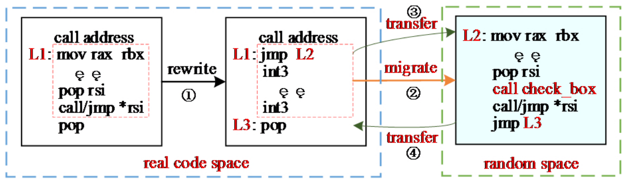

Figure 4: Migrate a code block to a random space

Code blocks containing ICT instructions collected by LBR should be judged by MProbe according to the security strategies (described later). The code block judged to be illegal is the probed code that can be used as a gadget. We will describe how MProbe prevents them from being used as gadgets later.

When Vector 3 is perceived because of *system trap*, MProbe locates the probed code block based on the offset between the exception address (the memory address where the EPT exception occurred) and the original code space. For example, assuming the exception address is *V*, the offset between it and the real code space is 1GB, and the probed code block is *V-1GB*. Due to the fine-grained ASLR, an attacker cannot deduce the addresses of other code blocks from the probed one. Therefore, the probed code block is the attacker's target, which is an available gadget if there is an ICT instruction in the code block.

Whether it is in Vector 2 or Vector 3, the probed code block containing a gadget will be migrated to a new random space, as shown in Figure 4. The first instruction of the code block in the real code space is rewritten as *jmp L2*, which transfers the control flow to the random space. Except for the first instruction, other instructions in the code block are rewritten as the one-byte instructions *int3*. So, unless the control flow is passed to *L2*, other calls to the probed code block will trigger a *system trap* because of executing *int3*, which can be captured by MProbe.

In the random space, the migrated code block is basically the same as the original one. We only add an instruction *jmp check_box* before the last instruction (ICT instruction) of the code block. *check_box* is a verification function, which can check the ICT instruction's legitimacy according to the security strategies. If it is legal, *check_box* will transfer the control flow back to the ICT instruction in random space. Otherwise, *check_box* executes the instruction *vmcall* to trigger a *system trap*. After that, MProbe will directly kill the current process. In addition, *check_box* uses LBR to get the executed code blocks and analyze their control flow transfer according to the security strategies.

There are two strategy sets established by MProbe, one is the constraint strategy set and the other is the protection strategy set. The former is used to ensure the probed code block to transfer the control flow to a legal location, and the latter prevents the probed code block from being called maliciously. The constraint strategies of the probed code blocks include the following:

A. *jmp ** only allows control flow jumping to the inside of the current function, and *call ** can only jump to the head of other functions. It should be noted that *longjmp* can be gained by parsing the *longjmp()* function in the ELF file, and we allow it to jump to the target address.
B. If without going through PLT, *call* and *jmp* cannot transfer the control flow to a library from application code, nor can transfer it to any other libraries from the current library.
C. The jump targets of ICT instructions must conform to the code alignment forms in the ELF file.
D. The return address of the instruction *ret* cannot be changed before *ret* is executed. To protect the return address of *ret*, MProbe needs to get the return address first. When there is a *ret* in the probed code block, we detect the code block recorded by the LBR to find the recently executed instruction *call*. If there is no *call*, the first code block recorded by LBR will be marked with *int3*. When the marked code block is executed again, we use LBR to find other 15 code blocks that have been executed. Follow this method and go on until the *call* that is paired with *ret* is located. After that, the instruction *call* will be redirected to a new code block, which can record the currently stored return address. Finally, when *ret* is executed, the recorded return address will be compared with the current one to check whether it has been tampered with.

The protection strategies are used to detect the control flow transferred to the probed code blocks, as shown below:

A. The instruction calling the probed code block in the real code space must conform to the constraint strategies.
B. If the instruction that transfers control flow to the probed code block in the real code space is jmp/call address, the transfer activity is legal.
C. If the instruction (i.e., *jmp L2* in Figure 4) that transfers control flow to the random space is in the probed code block, MProbe will look up which instruction transfers the control flow to the probed code block based on the LBR. The control flow transfer must conform to (1) and (2). It should be noted that, due to *MSR_LBR_SELECT* is in use, LBR only records ICT instructions and *call addresse*. So, if the control flow transfer instruction is *jmp address*, MProbe will not be able to locate it. When MProbe finds that the instruction recorded by LBR cannot jump to the probed code block in the real code space, we can judge that the control flow transfer instruction is *jmp address*, which is legal.
D. Any instruction must exploit the code at *L1* in Figure 4 to transfer control flow to random space. Otherwise, it is illegal.
E. Regardless of whether the current control flow violates the constraint strategies or the protection strategies, it will be judged to be illegal. After that, MProbe will start from the recently executed code block recorded by LBR for legitimacy detection. It uses both the constraint strategies and protection strategies to detect the legitimacy of the control flow transfer one by one, until a legal transfer activity is found.

In short, under the protection of MProbe, all the control flow transferred to the probed code block must be detected and analyzed. If it is illegal, MProbe can detect its illegal activities and prevent it from being used as a gadget.

6 TRANSFER LEGAL CONTROL FLOW

If a code block has been probed, it will be rewritten in the original space. Then, the probed code block will be migrated to a random

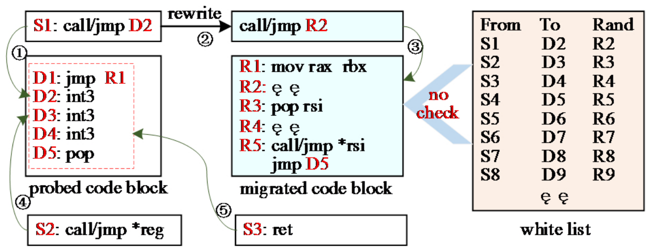

Figure 5: Transfer the legal control flow

space. For application code, we can directly modify its binary code in the host mode without affecting other processes. For the library code shared by multiple processes, the modification activity will affect the normal execution of other processes. First, if the code being rewritten is just called by other processes, an execution conflict will be triggered. Second, the random space is only mapped to the address space of the current process, not the address space of all processes. When other processes call the rewritten code block, the control flow will be redirected to an unmapped or wrong area, which will also cause exceptions.

To solve this problem, MProbe reallocates a new physical page for the probed library code , as shown in Figure 5. The memory page where the probed code block is located will be completely copied into the new page. In the new page, the probed code block will be rewritten, and the code on the original page remains unchanged. Then, MProbe modifies the last level page table of the current process, so that the table item pointing to the probed page points to the new page. After that, the current process's access to the probed code will be redirected to the new page without affecting other process's access to the probed code block in the original page. It should be noted that the addresses of the same library function in different processes are not the same. Therefore, attackers cannot use the library code probed in different processes to build a gadget chain.

The control flow transferred to the rewritten code in the real code space triggers a *system trap*, even it's legal. Although we can redirect the control flow to the random space after the *system trap* occurs, frequent *system traps* and security checking are expensive. In fact, if we can redirect the legal control flow to random space without triggering a *system trap*, or have no checking on the legal control flow, the performance will be reduced.

We set a whitelist for each probed code block. The white list contains three types of data, the source addresses of the control flow, the destination addresses of the control flow in the probed code block, and the address in the random space that the control flow should be redirected. All the legal control flow transfer activities will be recorded in the whitelist. If the control flow transfer instruction is *call/jmp address*, the operand *address* will be modified to the corresponding address in the random space. Therefore, the control flow can jump directly to the target address without causing any *system trap*. If the transfer instruction is an ICT instruction, and its source address and target address are in the white list, it will be redirected to the random space, which does not need to be checked again. The whitelist can reduce unnecessary system traps and legitimacy detection, thereby reducing performance overhead.

7 EVALUATION

7.1 Experimental Environment

We conduct all experiments on a Linux server, which is equipped with two 10-core Intel Xeon silver CPUs and 64GB memory. The OS is Ubuntu16.04 with kernel 4.15.

7.2 Security analysis

In this section, we simulate arbitrary read with HeartBleed[42], arbitrary jump with BROP[16], side-channel probing with AnC[37], and data leakage with DOP[38]. At the same time, we run MProbe to verify its defense capabilities. To ensure the attacks can be successfully deployed, we assume that the attacker has obtained the code segment scope (instead of accurate code addresses) through allocation oracles or */proc/pid/maps*.

Defense arbitrary read. We deploy HeartBleed in *openssl-1.0.1c*. HeartBleed triggers a *SIGSEGV* when reading unmapped memory, which is captured by MProbe. At this point, the attacker is blocked by MProbe before they can obtain the code content.

To further observe MProbe's response to code reading, we manually modify the source code *memcpy(bp,pl,payload)* in *openssl* to make *pl* point to the code segment. This attack scenario is possible in practice. If an attacker can handle the signal *SIGSEGV* itself, it can continuously increase *pl* without causing process crash until *pl* points to the code segment. When the code is read, HeartBleed will trigger an EPT exception. After that, MProbe blocks the current code reading activity.

Defense arbitrary jump. When deploying BROP[16], we skipped the step of cracking the canary and made BROP tamper with the return address directly. In our tests, BROP increase the return address byte by byte to search the available gadgets. The results show that the probability of triggering the signal *SIGILL* and *SIGSEGV* is more than 90% when BROP executes the first code block, and the probability is more than 99% when three code blocks are called continuously. If the attacker customizes the signal handler, it will be detected by MProbe when the *SIGSEGV* or *SIGILL* is triggered for the first time. If the attacker does not customize the signal handler, but restarts the process to perform code probing again after the process crashes, it will be detected in the second code probing. After that, it can also be detected and blocked by MProbe when probing code.

Defense side-channel probing. When AnC[37] is detected, the attacker has already cracked most bits of the target virtual address. At this time, only the last 3 bits of the index value of the page table at all levels are unknown. To crack the remaining bits, the attacker needs to access the memory at distances *n*4KB*, *n*2MB*, *n*1GB*, and *n*512GB* from the target address (n<8). We found that it is rare that the process code size exceeds 1GB, and there are almost no processes that exceed 512GB. MProbe maps the unmapped areas and make them inaccessible. Therefore, when an attacker accesses these areas, it will be perceived and blocked by MProbe, which leads the attacker cannot obtain the 30th~32nd bits and 39th~41st bits of the virtual address.

Furthermore, the typical side-channel attacks such as flush+reload[36], EVICT+TIME[44] and PRIME+PROBE[45] will be detected and blocked by MProbe whenever they read the code.

Table 1: CRAs defense results of MProbe

binary code	size	total gadgets	gadget chains	defense
libcodeblocks.so	4267	535758	70	√
libcapstone.so	869	109538	3	√
libfam.so	15	1969	1	√
libnetpbm.so.10	60	7704	1	√
libwxsmithlib.so	1719	187992	48	√
400.perlbench	877	100750	5	√
401.bzip2	45	3942	1	√
403.gcc	2285	254156	29	√
429.mcf	8	1079	1	√
471.omnetpp	401	56954	2	√

Defense data leak. We find DOP will be perceived and blocked by MProbe when it tries to read the GOT address stored in the PLT. Because the PLT is in the process code segment, and it is unreadable. In fact, even if an attacker is able to read the data stored in the GOT, it cannot obtain the real function addresses. The reason is that the library function addresses stored in the GOT has been transferred to the *secret code*. Although attackers can get the address of the *secret code*, they still cannot get the library function addresses because the *secret code* is unreadable.

Defense JIT-ROP. Currently, JIT-ROP has two types of attack forms, one requires code reading[43], and the other does not require code reading[35]. For the former, MProbe can detect and block it. For the latter, it turns the immediate into gadgets through the non-alignment feature of the code. In fact, this attack does not perform code probing, and it designs the gadgets to be used in advance. Therefore, it is not within the protection range of MProbe.

Defense other probing technologies. For arbitrary writes, when it triggers the signal *SIGSGEV*, it will be captured and blocked by MProbe. For process cloning, it can also be detected when combined with other probing techniques, such as arbitrary jump. However, MProbe cannot directly detect allocation oracles[48].

To verify the defense effect of MProbe against CRAs, we use ROPgadget[46] to search for available gadgets and manually connect them together to form a gadget chain. To achieve this goal, we add an execution breakpoint at the available gadget. When the gadget is executed, we pass the next gadget's address to the ICT instruction in the current gadget to simulate control flow hijacking. The defense results of MProbe are shown in Table 1. Each application in Table 1 contains at least one complete gadget chain.

We found that MProbe can detect the illegal control flow. The reason is the illegal control flow transfer violates the control flow constraint strategies and protection strategies formulated by MProbe. For example, the gadget in *libfam.so* needs to rewrite the return address, which will be detected and prevented by MProbe.

Even if attackers know the existence of MProbe, they cannot bypass MProbe. Because MProbe is built based on the attack nature. It analyzes malicious behavior from the perspective of underlying resource access and running trajectories. As long as the basic attack principles (such as code reading) employed by attackers remain unchanged, they can be detected.

Additionally, we found no false positives while monitoring *Apache*, *Redis*, and *Nginx*. Because, in normal execution scenarios, the process does not probe its own code or library code. As a result, MProbe will not actively track and analyze the control flow of the process. However, MProbe is ineffective for the CRAs that can be deployed without code probing, such as the CRAs deployed in unrandomized libraries.

7.3 Performance analysis

We use *SpecCPU2006* to test MProbe's impact on applications, and use *Lmbench* to test the system delay and bandwidth loss. No code probing occurred during the entire test. The results are shown in Figure 6 and Figure 7.

We can see that, the overhead introduced by MProbe is not too high when there are no probing attacks. The reason is MProbe is a passive security solution. It tracks and analyzes control flow if and only if there exist abnormal activities. In most scenarios, there is no abnormal activity. As a result, the normal processes will not be affected too much, and the overhead introduced by MProbe is not too high.

Compared with the native OS, MProbe introduces new execution modes, namely *guest* and *host*. The OS switches between the two modes, which causes *system traps*. To further observe the impact of MProbe on OS, we use some micro benchmarks to test the running overhead, as shown in Table 2. The results show that, the *system trap* is a main factor affecting OS performance. For some legal instructions (such as *jmp address*) related with the probed code block, MProbe adds them to the white list. Therefore, they will only trigger a system trap when executed for the first time, and won't trigger system traps in the subsequent execution.

The *system traps* can be divided into two: unconditional traps and conditional traps. In the *guest*, the execution of the instructions *CPUID, GETTSEC, INVD, XSETBV* and all VMX instructions except *VMFUNC* will cause the OS to unconditionally fall to the *host* from the *guest*. Conditional traps are triggered by specific events, such as EPT exceptions. Before the probing attack occurs, MProbe will not actively track any control flow. Therefore, the frequency of conditional traps is very small in a normal execution scenario. Another word, the overhead caused by MProbe in normal execution scenarios mainly comes from the unconditional traps triggered by specific instructions. We found that, among all the instructions

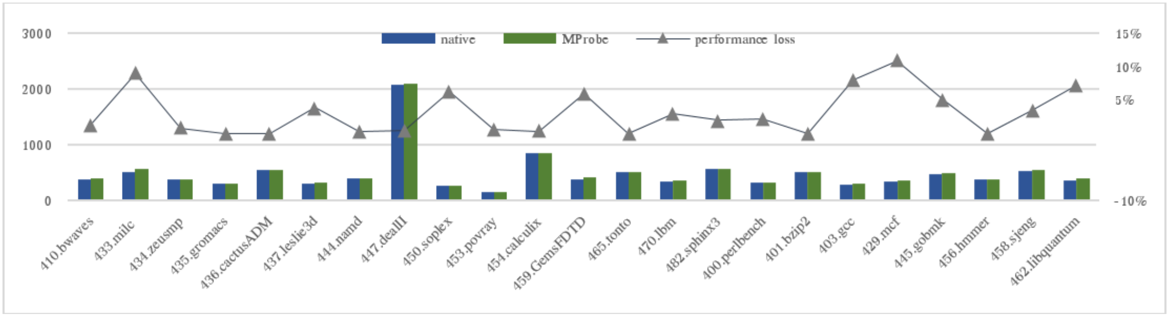

Figure 6: SpecCPU2006 test results. The abscissa is the benchmark. The ordinate on the left is the basic running time, which corresponds to the bar graph; the ordinate on the right is the performance degradation factor, which corresponds to the line graph. The maximum speed degradation factor of each test is less than 10%, and the average degradation is less than 3%.

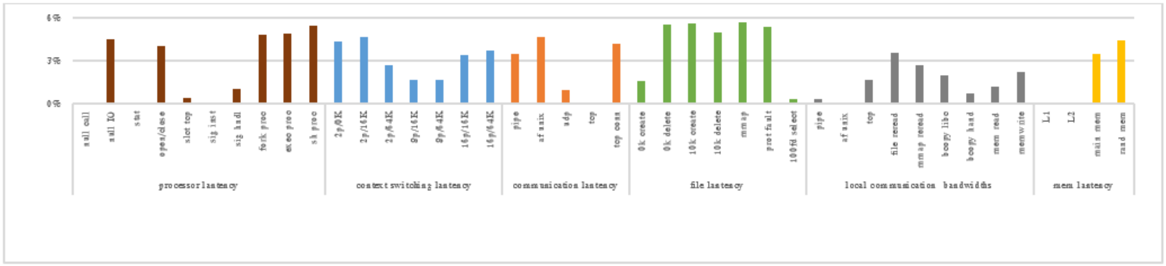

Figure 7: Lmbench test results. The abscissa is the benchmark, and the ordinate is the performance degradation factor. The average performance loss of each group of tests from left to right is 2.5%, 3.1%, 2.6%, 4.1%, 1.6%, 2%, and the average of all test items is 2.6%.

Table 2: The test results of micro benchmarks (ns). call probed code: CP; library call: LC; jump to probed code: JP; return to probed code: RP

No MProbe						Running MProbe							
call	LC	ret	jmp	ept switch	system trap	1st CP	2nd CP	1st LC	2nd LC	1st RP	2nd RP	1st JP	2nd JP
2.88	3.39	2.57	2.06	119.06	524.35	908.05	2.95	1425.39	3.4	1009.15	69.73	964.43	2.11

causing unconditional traps, the *CPUID* execution frequency was much higher than other instructions, which is a key factor introducing overhead. For some legal instructions (such as *jmp address*) related with the probed code block, MProbe adds them to the white list. Therefore, they only trigger a system trap when executed for the first time. When they are executed again, they jump directly to the correct address without causing any system traps. Therefore, they do not introduce too much overhead.

In order to test the impact of MProbe on the probed process after the probing attack occurs, we read the binary code of *SpecCPU2006* through a kernel thread to simulate a probing attack. To test the impact of the size of the probed code on MProbe, we continuously increase the percentage of probed code. After that, we test the impact of MProbe on the running speed of the probed process. The test results are shown in Figure 8.

Obviously, MProbe has a greater impact on the running speed of the probed process. The more probed code, the more MProbe affects the process. After code probing occurs, the OS will fall into *host*. MProbe creates a random space for the probed code. Then, MProbe will track and analyze the control flow until it can detect the control flow's legitimacy.

During the detection, the OS will switch between the *guest* and *host* multiple times, which will cause multiple system traps. In the *host*, the process will be suspended. After that, MProbe will analyze the legitimacy of the control flow. When the OS returns to the guest, the process is woken up again. The more code that is probed, the more code is tracked and analyzed, which inevitably results in longer process hang times. Therefore, the running speed of the process will decrease as the size of probed code increases.

Fortunately, the code probing can happen if and only if the OS is invaded. In most cases, there is no code probing in the OS. Taking a step back, once a code probing occurs, it is worth sacrificing about 20% of the execution speed of the probed process in exchange for the

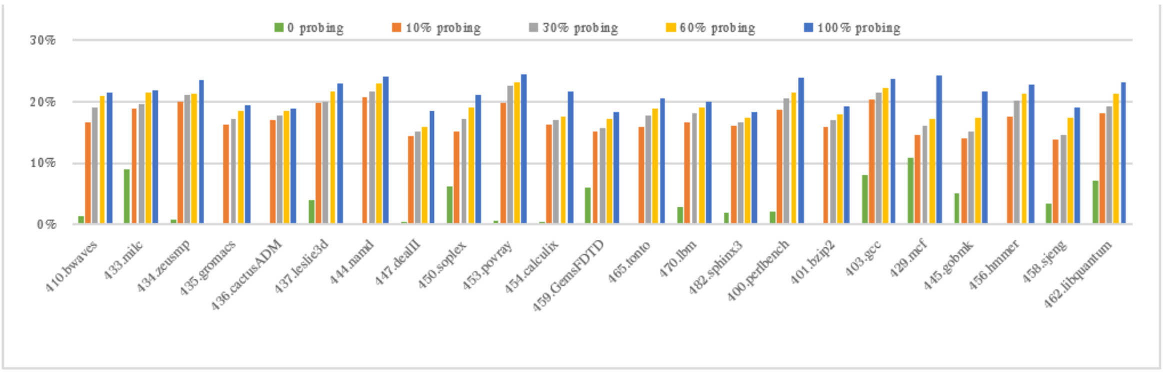

Figure 8: MProbe's impact on the process after a probing attack. After the probing occurs, the process speed slowdowns by about 15%~25%. The execution speed of the process will increase as the number of probed code increases.

security of the entire OS. Therefore, we believe that the performance loss introduced by MProbe to the system is acceptable.

Similar to BUDDY[47], we also tested the impact of MProbe on the real application Apache. The results are shown in Appendix A. In addition, we also compare MProbe with existing security methods, which is shown in Appendix B.

8 CONCLUSIONS

To mitigate the CRAs based on code probing, this paper proposes an anti-probing attack method MProbe. It builds a series of anti-probing mechanisms to perceive and prevent attackers' code probing activities. When MProbe perceives code probing, it creates a random space for the probed code and cancel the probed code's execution permission in the native address space. Therefore, although an attacker can probe code content and address in the native address space, it cannot execute the probed code. The results show that MProbe introduces less than 3% performance overhead to the OS.

However, MProbe still has some limitations. First, it is only effective for the code probing in user space, but not effective in kernel space. Since the kernel resides in memory for a long time and is completely shared, any modification to the kernel will affect all execution entities. This will undoubtedly greatly increase the performance of the OS. In addition, the kernel itself has certain self-debugging capabilities, and it needs to have code reading capabilities. For example, the kernel will read the code that caused kernel errors. It is currently difficult for us to efficiently distinguish between legal code reading and illegal code reading in kernel. Second, MProbe can only be deployed on x86 architecture processors. Third, MProbe is only valid for open source Linux, but invalid for closed source Windows. How to improve MProbe's universality in different software and hardware will be a focus of our future work.

REFERENCES

[1] Biondo A, Conti M, *et al.* "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel {SGX}". *Proc. The 27th USENIX Security Symposium.* 2018: 1213-1227.
[2] He W, Das S, Zhang W, *et al.* "BBB-CFI: lightweight CFI approach against code-reuse attacks using basic block information". *ACM Transactions on Embedded Computing Systems*, 2020, 19(1): 1-22. DOI:https://doi.org/10.1145/3371151
[3] Crane S J, Volckaert S, Schuster F, *et al.* "It's a TRaP: Table randomization and protection against function-reuse attacks". *Proc. The 22nd ACM SIGSAC Conference on Computer and Communications Security.* 2015: 243-255. DOI:https://doi.org/10.1145/2810103.2813682
[4] Kayaalp M, Schmitt T, Nomani J, *et al.* "Signature-based protection from code reuse attacks". *IEEE Transactions on Computers*, 2013, 64(2): 533-546. DOI:https://doi.org/10.1109/tc.2013.230
[5] DeLozier C, Lakshminarayanan K, Pokam G, *et al.* "Hurdle: Securing Jump Instructions Against Code Reuse Attacks". *Proc. The 25th International Conference on Architectural Support for Programming Languages and Operating Systems.* 2020: 653-666. DOI:https://doi.org/10.1145/3373376.3378506
[6] Dang T H Y, Maniatis P, *et al.* "The performance cost of shadow stacks and stack canaries". *Proc. The 10th ACM Symposium on Information, Computer and Communications Security.* 2015: 555-566.DOI: https://doi.org/10.1145/2714576.2714635
[7] Marco-Gisbert H, Ripoll. "Address space layout randomization next generation". *Applied Sciences*, 2019, 9(14): 2928. DOI:https://doi.org/10.3390/app9142928
[8] Wang C, Chen B, Liu Y, *et al.* "Layered object-oriented programming: Advanced vtable reuse attacks on binary-level defense". *IEEE Transactions on Information Forensics and Security*, 2018, 14(3): 693-708. DOI:https://doi.org/10.1109/tifs.2018.2855648
[9] Ho J W. "Efficient and robust detection of code-reuse attacks through probabilistic packet inspection in industrial IoT devices". *IEEE Access*, 2018, 6: 54343-54354. DOI:https://doi.org/10.1109/access.2018.2872044
[10] D. Williams-King, G. Gobieski, K. Williams-King, *et al.* "Shuffler: Fast and deployable continuous code re-randomization," *Proc. The OSDI*, 2016.
[11] Chao Zhang, Tao Wei, *et al.* "Practical Control Flow Integrity and Randomization for Binary Executables". *Proc. 2013 IEEE Symposium on Security and Privacy.* Washington, DC, USA, 559–573. DOI:https://doi.org/10.1109/sp.2013.44
[12] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection". *Proc. The 23rd USENIX Conference on Security Symposium.* Berkeley, CA, USA, 401–416.
[13] Gupta A, Kerr S, Kirkpatrick M S, *et al.* "Marlin: A fine grained randomization approach to defend against ROP attacks". *Proc. The International Conference on Network and System Security.* Springer, Berlin, Heidelberg, 2013: 293-306. DOI:https://doi.org/10.1007/978-3-642-38631-2_22
[14] Hiser J, Nguyen-Tuong A, *et al.* "ILR: Where'd my gadgets go?" *Proc. The IEEE Symposium on Security and Privacy.* IEEE, 2012: 571-585. DOI:https://doi.org/10.1109/sp.2012.39
[15] Lu†Kangjie, Nürnberger S, Backes M, *et al.* "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization". *Proc. The 23rd Network and Distributed System Security Symposium* (NDSS), San Diego, CA, USA, 2016. DOI:https://doi.org/10.14722/ndss.2016.23173
[16] Bittau, A., Belay, A., *et al.* "Hacking Blind". *Proc. The IEEE Security and Privacy*, 2014, 227–242. DOI:https://doi.org/10.1109/sp.2014.22
[17] Rudd R, Skowyra R, Bigelow D, *et al.* "Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity." *Proc. NDSS.* 2017.
[18] Gktasgktas E, Gawlik R, *et al.* "Undermining Information Hiding (And What to do About it)". *Proc. The 25th USENIX Security.* 2016.
[19] Gawlik R, Kollenda B, Koppe P, *et al.* "Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding". *Proc. The NDSS.* 2016, 16: 21-24. DOI:https://doi.org/10.14722/ndss.2016.23262
[20] Göktas E, Razavi K, *et al.* "Speculative Probing: Hacking Blind in the Spectre Era". *Proc. The 2020 ACM SIGSAC Conference on Computer and Communications Security.* 2020: 1871-1885. DOI:https://doi.org/10.1145/3372297.3417289
[21] Braden K, Davi L, *et al.* "Leakage-Resilient Layout Randomization for Mobile Devices". *Proc. The NDSS.* 2016, 16: 21-24. DOI:https://doi.org/10.14722/ndss.2016.

23364
[22] Chen X, Bos H, Giuffrida C. "CodeArmor: Virtualizing the code space to counter disclosure attacks". *Proc. The IEEE European Symposium on Security and Privacy,* 2017: 514-529. DOI:https://doi.org/10.1109/eurosp.2017.17
[23] Kuznetsov V, Szekeres, László, Payer M, *et al.* "Code-Pointer Integrity". *Proc. The Usenix Symposium on Operating Systems Design & Implementation.* 2014. DOI:https://doi.org/10.1145/3129743.3129748
[24] Lu K, Song C, Lee B, *et al.* "ASLR-Guard: Stopping address space leakage for code reuse attacks". *Proc. The 22nd ACM SIGSAC conference on computer and communications security.* 2015: 280-291. DOI:https://doi.org/10.1145/2810103.2813694
[25] Davi L, Liebchen C, Sadeghi A R, *et al.* "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming". *Proc. The Network and Distributed System Security Symposium.* 2015. DOI:https://doi.org/10.14722/ndss.2015.23262
[26] Backes M, Nürnberger S. "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing". *Proc. The 23rd USENIX Security Symposium.* 2014: 433-447.
[27] HongHu, ChenxiongQian, *et al.* "Enforcing Unique Code Target Property for Control-Flow Integrity". *Proc. The 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, New York, NY, USA, 1470–1486. DOI:https://doi.org/10.1145/3243734.3243797
[28] Caroline Tice, Tom Roeder, *et al.* "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM". *Proc. The 23rd USENIX Conference on Security Symposium.* USENIX Association, Berkeley, CA, USA, 941–955.
[29] Ben Niu and Gang Tan. "Per-Input Control-Flow Integrity". *Proc. The 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, NY, USA, 914–926. DOI:https://doi.org/10.1145/2810103.2813644
[30] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, *et al.* "VTint: Protecting Virtual Function Tables' Integrity". *Proc. The Network and Distributed System Security Symposium,* 2015. DOI:https://doi.org/10.14722/ndss.2015.23099
[31] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, *et al.* "VTrust: Regaining Trust on Virtual Calls". *Proc. The Network and Distributed System Security Symposium,* 2016. DOI:https://doi.org/10.14722/ndss.2016.23164
[32] Criswell J, Dautenhahn N. "KCoFI: Complete control-flow integrity for commodity operating system kernels." *Proc.* The *IEEE Symposium on Security and Privacy.* IEEE, 2014: 292-307. DOI:https://doi.org/10.1109/sp.2014.26
[33] Zhang M, Sekar R. "Control flow integrity for {COTS} binaries" Proc. *22nd USENIX Security Symposium.* 2013: 337-352. https://doi.org/10.1145/2818000.2818016
[34] Mohan V, Larsen P, Brunthaler S, *et al.* "Opaque Control-Flow Integrity" *Proc. NDSS.* 2015, 26: 27-30. DOI:https://doi.org/10.14722/ndss.2015.23271
[35] Maisuradze G, Backes M, *et al.* "What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses", *Proc. 25th USENIX Security Symposium,* 2016: 139-156.
[36] Yarom Y, Falkner K. "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack" *Proc. 23rd USENIX Security Symposium.* 2014: 719-732.
[37] Gras B, Razavi K, Bosman E, *et al.* "ASLR on the Line: Practical Cache Attacks on the MMU" *Proc. NDSS.* 2017, 17: 26. DOI:https://doi.org/10.14722/ndss.2017.23271
[38] Hu H, Shinde S, Adrian S, *et al.* "Data-oriented programming: On the expressiveness of non-control data attacks". *Proc. IEEE Symposium on Security and Privacy (SP).* 2016: 969-986. DOI:https://doi.org/10.1109/sp.2016.62
[39] Hu Z, Chen P, *et al.* "A co-design adaptive defense scheme with bounded security damages against Heartbleed-like attacks". *IEEE Transactions on Information Forensics and Security,* 2021, 16: 4691-4704. DOI:https://doi.org/10.1109/tifs.2021.3113512
[40] Omotosho A, Welearegai G B, Hammer C. "Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters". *Proc.* the *37th ACM/SIGAPP Symposium on Applied Computing.* 2022: 510-519. DOI:https://doi.org/10.1145/3477314.3507108
[41] Lin K, Xia H, Zhang K, *et al.* "AddrArmor: An Address-based Runtime Code-reuse Attack Mitigation for Shared Objects at the Binary-level." *Proc. 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom).* 2021: 117-124. DOI:https://doi.org/10.1109/ispa-bdcloud-socialcom-sustaincom52081.2021.00029
[42] Wang J, Zhao M, Zeng Q, *et al.* "Risk assessment of buffer" Heartbleed" over-read vulnerabilities." *Proc. 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 2015: 555-562. DOI:https://doi.org/10.1109/dsn.2015.59
[43] Snow K Z, Monrose F, Davi L, *et al.* "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization." *Proc. IEEE Symposium on Security and Privacy.* 2013: 574-588. DOI:https://doi.org/10.1109/sp.2013.45
[44] Osvik D A, Shamir A, Tromer E. "Cache attacks and countermeasures: the case of AES." *Proc. Cryptographers' track at the RSA conference.* 2006: 1-20. DOI:https://doi.org/10.1007/11605805_1
[45] Liu, Fangfei, *et al.* "Last-level cache side-channel attacks are practical." *Proc. IEEE symposium on security and privacy.* 2015.
[46] J. Salwan. "ROPgadget–Gadgets Finder and Auto-Roper." http://shell-storm.org/project/ROPgadget
[47] Lu K, Xu M, Song C, *et al.* "Stopping memory disclosures via diversification and replicated execution." *IEEE Transactions on Dependable and Secure Computing,* 2018, 18(1): 160-173. DOI:https://doi.org/10.1109/TDSC.2018.2878234
[48] Oikonomopoulos A, Athanasopoulos E, Bos H, *et al.* "Poking holes in information hiding." *Proc. USENIX Security Symposium (USENIX Security 16).* 2016: 121-138.
[49] Jan Werner, George Baltas, Rob Dallara, Nathan Otternes, Kevin Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS).
[50] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In CCS.
[51] Zhang M, Polychronakis M, Sekar R. Protecting cots binaries from disclosure-guided code reuse attacks[C]//Proceedings of the 33rd Annual Computer Security Applications Conference. 2017: 128-140.
[52] Tang A, Sethumadhavan S, Stolfo S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads[C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 256-267.
[53] Crane S, Liebchen C, Homescu A, *et al.* Readactor: Practical code randomization resilient to memory disclosure[C]//2015 IEEE Symposium on Security and Privacy. IEEE, 2015: 763-780.
[54] Chen Y, Zhang D, Wang R, *et al.* NORAX: Enabling execute-only memory for COTS binaries on AArch64[C]//2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017: 304-319.
[55] Gionta J, Enck W, Ning P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities[C]//Proceedings of the 5th ACM Conference on Data and Application Security and Privacy. 2015: 325-336.
[56] Bigelow D, Hobson T, Rudd R, *et al.* Timely rerandomization for mitigating memory disclosures[C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 268-279.

APPENDIX A

The results of testing Apache are shown as Table3, which show that MProbe slowdowns Apache by an average of 4.29%, which indicates MProbe is capable of handling multi-process programs efficiently.

APPENDIX B

Compared with existing methods, MProbe has defense effects on more code probing technologies, which is shown as Table 4. Readactor[53] and TASLR[56] rely on the source code, causing them to invalidate the inline assembly or extra code introduced by the linker and loader. Secret[51] remaps all code pointers. If a pointer is rewritten arbitrarily, it will be detected due to no matched pointer with it. Therefore, Secret has a certain defense effect on arbitrary write and arbitrary jump. However, it is difficult and expensive to obtain and manipulate all pointers. XnR[50] will switch the sliding window due to frequent code jumps, which will cause huge overhead in corner cases. Most of the methods in Table 4, such as Heisenbyte[52], NORAX[54], and HideM[55], do not mention the defense to process cloning. In addition, except for MProbe, other methods are invalid to side channel probing attacks. Moreover, because MProbe does not actively track all control flow transfers, it will not introduce excessive overhead.

Table 3: Overhead of processing time (ms) per request incurred to Apache httpd under MProbe with different numbers of worker processes and different numbers of concurrent connections. s=1MB.

Level	c=1			c=16			c=64			c=128			c=256		
worker	Orig.	MProbe	Loss	Orig.	MProbe	Loss	Orig.	MProbe	Loss	Orig.	MProbe	Loss	Orig.	MProbe	Loss
p=1	17.3	17.6	1.73%	14.8	15.1	2.03%	13.6	13.9	2.21%	12.9	13.5	4.65%	13.1	13.8	5.34%
p=2	16.5	17.5	6.06%	12.7	13.6	7.09%	11.1	11.5	3.60%	12.5	13.1	4.80%	12.8	13.4	4.69%
p=3	18.1	18.4	1.66%	11.9	12.8	7.56%	9.9	10.8	9.09%	12.1	12.4	2.48%	13.2	13.9	5.30%
p=4	17.4	17.9	2.87%	11.4	12.9	13.16%	10.2	10.3	0.98%	12.6	12.9	2.38%	14.1	14.1	0.00%
p=5	17.2	18.5	7.56%	10.3	10.6	2.91%	9.8	10.5	7.14%	11.9	12.2	2.52%	12.9	13.5	4.65%
p=6	16.9	17.6	4.14%	11.1	11.9	7.21%	9.6	9.9	3.13%	11.8	12.1	2.54%	13.4	13.4	0.00%
p=7	17.2	18.2	5.81%	10.4	11.2	7.69%	10	10	0.00%	12.2	12.5	2.46%	13.2	13.3	0.76%
p=8	16.8	18.5	10.12%	10.2	11.5	12.75%	10.1	10.2	0.99%	13.1	13.1	0.00%	14.3	14.5	1.40%
Geomean			4.99%			7.55%			3.39%			2.73%			2.77%

Table 4: Comparison with existing methods. NSC: not rely on source code. M-overhead: maximum overhead in SPEC. *: effective for part.

APP	NSC	arbitrary read	arbitrary write	arbitrary jump	side channel	data leak	process clone	M-overhead
Secret[51]	√	√	*	*	x	√	x	36%
NEAR[49]	√	√	x	x	x	*	x	19.88%
Heisenbyte[52]	√	√	x	x	x	*	x	62%
XnR[50]	√	√	x	x	x	*	x	526%
Readactor[53]	x	√	x	x	x	x	x	26%
NORAX[54]	√	√	x	x	x	√	x	-
HideM[55]	√	√	x	x	x	*	x	<7%
TASLR[56]	x	*	*	x	x	x	√	10.1%
MProbe	√	√	*	√	√	√	*	<15%