Future Generation Computer Systems 140 (2023) 282–298

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

MagBox: Keep the risk functions running safely in a magic box

YongGang Li [a,*], GuoYuan Lin [a], Yeh-Ching Chung [b], YaoWen Ma [a], Yi Lu [a], Yu Bao [a]

[a] The School of Computer Science and Technology in the China University of Mining and Technology, Xuzhou, Jiangsu, 221116, PR China
[b] The School of Data Science, CUHK(SZ), Shenzhen, Guangdong, 518172, PR China

ARTICLE INFO

Article history:
Received 19 July 2022
Received in revised form 16 October 2022
Accepted 31 October 2022
Available online 4 November 2022

Keywords:
Integrity
System architecture
Security and protection
Software and system safety

ABSTRACT

Address space layout randomization (ASLR) has been widely deployed in operating systems (OS) to hide memory layout, which mitigates code reuse attacks (CRAs). Unfortunately, the memory probing techniques can still provide attackers with enough information to bypass ASLR. Although the control flow integrity (CFI) methods are not affected by code probing, they face the precision problem of control flow graphs (CFG). To make matters worse, most methods rely on the source code of the targets to be protected, which leads to their restrictions on the protection of the objects without source code. To solve these problems, MagBox is proposed in this paper. It identifies the risk functions that can provide gadgets for CRAs by detecting and analyzing attackers' code probing activities. If the function is probed, it will be moved to a new address space. After that, the control flow transfers of the function will be tracked and analyzed in real time to judge their legitimacy. Experiment results and analysis show that MagBox can mitigate CRAs, and only introduces 3.4% performance overhead to the CPU.

1. Introduction

CRAs can be used to hijack the control flow. ROP (Return-Oriented Programming) and JOP (Jump-Oriented Programming) are two typical attacks in CRAs [1]. Moreover, attackers have developed a variety of variants, such as tiny-jop [2], LOP [3], and CALL-ROP [4], etc. These attacks are adjusted to fight against the existed defense models, making them lose their detection and defense effects.

ASLR [5], especially the fine-grained ASLR, changes the memory layout, which makes the attacker unable to get gadgets. Unfortunately, attackers have developed a variety of probing techniques including allocation oracle, arbitrary write, arbitrary read, and arbitrary jump [6], etc. Based on these techniques, attackers can still obtain the available gadgets [7]. Attackers can also get the hidden code address through process cloning [8] and side-channel attacks [9].

The CRA defense methods based on CFI does not pay attention on the code addresses [10]. They only care about whether the control flow path is legal. The static CFI analyzes the source code of the application to construct a CFG offline [11,12]. The ideal CFG covers all possible control flow paths of the application. However, the control flow paths cannot be accurately enumerated all the time due to the different execution conditions, such as the input. Worse, the CFG of the complex software will be so large that the state explosion problem is inevitable.

The dynamic CFI tracks and analyzes the control flow paths in real time, which can get rid of the dependence on static CFG [13]. In practice, there are many types of instructions that can transfer control flow (called control flow transfer instructions in this paper). To ensure that no jump branch is missed, the existed methods track all control flow transfer instructions. However, not all control flow transfer instructions can be used as gadgets, which leads to a lot of unnecessary tracking.

In addition, the current CFI methods (such as MCFI [14] and IFCC [15]) rely on the logical information (such as the control flow transfer condition) provided by the source code of the objects to be protected. Therefore, their protection effect on the objects (such as the loaded library code) without source code is very limited.

In summary, both ASLR and CFI have their limitations. ASLR can be bypassed by various memory probing technologies, and loses its protection effect. On the one hand, CFI relies on the source code, causing the protection failure to the library code. On the other hand, a lot of meaningless control flow tracking is introduced, which affects CFI's execution efficiency and protection accuracy.

Since the ASLR is widely used, memory probing has become a key step for deploying CRAs [16]. It can provide code addresses or code forms for attackers. Dur to the huge size of the available address space (as much as 128 TB for the user space), the area of an application occupies only a small part of the entire space. If there are no known gadgets, attackers need to search for the code snippets meeting gadget forms in the huge space. For example, BROP [7] modify the return address bite by bite to find the

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

https://doi.org/10.1016/j.future.2022.10.035
0167-739X/

instruction "*pop register; ret*". If a gadget is known but its address is unknown, the attacker needs to crack the address. For example, the side channel attack [9] uses the cache hits of the page tables at all levels to obtain the 12nd to 48th bits of the virtual address.

The probed code block that can transfer control flow to a non-fixed position will be selected by attackers to perform the malicious activities. Such code block contains an ICT (indirect control transfer) instruction (such as *ret, call *xx,* and *jmp *xx*) whose control data stores in a writable area. The function to which the probed ICT instruction belongs is called risk function in this paper.

If we look the memory probing from another light, it can help not only attackers but defenders as well. First, the memory probing activity can expose the attack intensions. Finding attackers' malicious attempts before the CRA deployment can help us take defense measures in advance. Second, the probed memory exposes the risk function that may be selected by an attacker. Finding the risk functions can allow us to track and detect the specific code that may be used as a gadget, instead of all the jump instructions. The MagBox proposed in this paper exploits the attacker's probing activities to discover risk functions and track their control flow transfers to judge whether they are called legally.

How to perceive the memory probing and identify the risk function selected by the attacker is the first challenge faced by MagBox. Under the ASLR protection, the existed attack technologies cannot get a complete gadget chain or all the accurate addresses of gadgets through a single memory probing. Multiple memory probing is unavoidable for attackers, which exposes the attacker's probing intention. There exist differences between the probing activities and the normal activities. These differences can be used to perceive attackers' memory probing.

For CRAs, only the code that can transfer control flow to a non-fixed position may be selected as a gadget. In theory, both the static method based on source code analysis and the dynamic method based on binary scanning according to specific code forms can actively detect the risk code. However, the risk code is selected by us, not attackers, whose probability of being used by an attacker is very low. If a probed code snippet can transfer the control flow to a non-fixed position, it is likely to be selected by the attacker. Therefore, we can identify the risk function through the probing activity and probing results of the attackers.

How to track the control flow of the risk function in real time and determine its legitimacy is another challenge faced by MagBox. In the native OS, we lack the ability to monitor and control the execution entity, which makes it difficult to track control flow in real time. To solve this problem, we must have the ability to monitor and control the behavior of execution entities. Then, we can track the activity of control flow transfer instructions and record the control flow paths of the risk function, which are the basis for formulating security strategies. Last, the legitimacy of the risk function can be judged based on the security strategies.

In summary, the contributions of MagBox are as follows:

(1) Build a model perceiving memory probing. The model creates a fake space mechanism that can passively perceives the memory probing activities and identify the rusk functions.
(2) Build a control flow tracking model at binary level. The risk function will be migrated to a new address space with a specific permission configuration. In the new address space, its control flow transfers will be monitored and analyzed.
(3) Build new security strategies for judging the legitimacy of risk functions. The historical control flow paths and the current instruction types will be combined to judge the risk function's legitimacy.

2. Related works

To defend against CRAs, researchers have conducted many researches. The current methods mainly focus on CFI protection and ASLR.

2.1. CFI protection

CRAs will cause abnormal instruction sequences in applications. For example, the ROP needs to use the instruction *ret* to connect all gadgets together, resulting in frequent *ret* in the control flow paths, while there is no instruction *call* in pairs with it. In [17], whether the *call* and *ret* appear in pairs is used as a criterion for identifying ROP. It can detect some ROP, but not effective on ROP variants (such as COOP [18]). DROP [19] believes that the maximum number of instructions is less than 5 in a single gadget, and the number of gadgets should be more than 3. Following the prerequisites set by DROP, SCRAP [20] defends against JOP by analyzing the attack feature logic and filtering out false function calls in the attack. However, tracing all the jump branches will significantly reduce the execution speed of the target process. kBouncer [21] first reads and analyzes the jump branches provided by LBR (Last Branch Recording) registers. Then it checks whether *ret* and *call* are stored in the memory pointed to by two adjacent registers. However, attackers can obfuscate the two adjacent LBR registers with invalid code, causing detection failure.

Other methods analyze the legitimacy of control flow paths to detect CRAs. πCFI [22] is a fine-grained method based on MCFI [14]. It maps the application code into two types (writable and non-writable) to gain the code write permission, which derives a high-risk attack surface and poses a threat because of the writeable code. CFIGuard [23] is a fully transparent CRA detection engine for user applications. It uses LBR and Performance Monitor Unit (PMU) to monitor every executed indirect branch during the lifetime of a process. Similar with kBouncer [21], the historical information of the LBR may be confused by the attacker, leading to detection failure. IFCC [15] guarantees CFI by modifying GCC and LLVM. But it is only valid for forward-edge. When compiling the application program, μCFI [24] identifies those instructions that may affect the jump branches and allows the program to record some necessary execution data. When the program is running, μCFI monitors it in another process, records environmental data on some key instructions, and judges whether the jump path meets expectations. The tracking accuracy of μCFI is very high, but it significantly reduces the execution speed of the process.

There exist more coarse-grained CFI methods including KCoFI [25], CCFIR [26], binCFI [27], O-CFI [28], and PAL [29] etc. The coarse-grained CFGs are easier to build, even without access to source code. But on the downside, the coarse-grained CFGs are too permissive so that it is still possible to mount attacks in general.

2.2. ASLR

ASLR changes the code layout in memory, so that an attacker cannot know the code address. Marlin [30] decomposes the binary file of the application into multiple code blocks with function as a granularity. Then it randomly distributes all the parts. Compared with Marlin, ILR [31] can randomize each instruction in a program. However, JIT-ROP [32] can bypass the two methods, leading to defense failure. Isomeron [33] combines the instruction paths and code addresses together for randomization. The problem is it only performs randomization once when the process is loaded. Due to the memory disclosure vulnerability [34], CloneProbing [8] can obtain the address by cloning the address space of the parent process.

To solve the clone problem, researchers perform periodic or real-time randomization. STABILIZER [35] periodically randomizes the code and stack addresses. However, its efficiency is affected by the length of the randomization cycle. Compared with STABILIZER, RUNTIMEASLR [8] only randomizes the address of the child process when the parent process is cloned. However, STABILIZER and RUNTIMEASLR have no defense against CRAs caused by memory leaks [36,37].

3. Assumptions and threat models

First, we assume the address spaces of execution entities have been randomized with the function granularity. Attackers must probe memory to find enough gadgets, which is similar with Buddy [16]. Now, ASLR has been widely adopted by the OS. The randomized objects cover multiple granularities such as pages, functions, code blocks, and even instructions. Code randomization at function granularity shuffles the distribution of all functions in memory. Therefore, the address disclosure of any one function will not expose the addresses of other functions. For the randomized code, attackers must probe code multiple times to obtain enough gadgets.

Second, we assume the application code will not be re-randomized when it restarts without re-compilation. In most scenarios, randomization happens during compilation. The memory layout is fixed after the code is compiled. In practice, randomizing all code in real time is nearly impossible. Application code can be re-randomized when the process crashes or restarts, but this cannot be applied to the loaded libraries it relies on. Because, changing the memory layout of the shared library will affect other running processes, thereby causing execution errors.

Third, we assume that attackers can probe the applications in user space. Current probing methods include allocation oracle, arbitrary write, arbitrary read, arbitrary jump, process cloning, side-channel attacks, and data leak, which have been proven to be feasible, such as [6,7,38,39].

Fourth, we assume that attackers cannot modify the application code. The code segment of all the processes will be mapped as non-writable, and any code writing will trigger a segment fault. Although the attacker can turn off the write protection of the code by manipulating the *cr0* register, *cr0* can be protected by the *cr0 shadow* in VMCS [40].

Fifth, we assume the attacked process can capture the signal (such as *SIGSEGV*) and respond to it. In practice, in addition to *SIGSTOP* and *SIGKILL*, the process can capture all other signals. Applications (such as *Nginx*) use this mechanism to prevent process crashes or restart the crashed process immediately. Based on this mechanism, an attacker can repeatedly probe the address space of a process.

Moreover, we do not consider attacks with privileged code execution abilities, and the memory layout cannot be obtained by reading */proc/pid/mem*. The threat model in this paper focuses on the following 7 types of memory probing.

Vector 1: Allocation Oracle (such as [38]). It uses memory allocation functions (such as *malloc* and *new*) to allocate an area to the process. Although allocation oracle cannot get the accurate address, it can know which area has been mapped to the process according to the return results.

Vector 2: Arbitrary Write (such as the vulnerability-based overflow). It can perform a write operation to arbitrary memory. On the one hand, arbitrary write can directly modify the control data in the writable memory (such as tampering with the return address), and on the other hand, it can also be used to probe the properties of the target memory. For example, if a segment fault generated when writing data to the mapped memory, the unwritable data or code must be there.

Vector 3: Process Cloning Probing [8]. It uses the cloned child process to probe the memory layout of the parent process, which can avoid the parent process's crash caused by illegal access.

Vector 4: Arbitrary Read (such as heart bleed [41]). It can directly read the content, including code and data, from arbitrary memory.

Vector 5: Data-Leak [42]. Attackers can exploit DOP [42] to read the relative offset in PLT (Procedure Linkage Table), and then they can get the randomized address of GOT (Global Offset Table), where stores all the library function addresses need by the current process.

Vector 6: Arbitrary Jump (such as BROP [7]). It can redirect the control flow to any position. Then the available gadgets can be located by analyzing to the crash information caused by the control flow transfer.

Vector 7: Side-channel Attack [9]. It uses the cache hits of the page tables at all levels to crack the 12th to 48th bits of the virtual address step by step.

4. Overall design of MagBox

The first challenge faced by MagBox is how to perceive attackers' memory probing. Any memory probing has its unique behavior characteristics. These characteristics are the keys that MagBox perceives memory probing.

In user space, only a small part of the address space is mapped to the process, and most of the remaining address space is unmapped. Therefore, the probing operation has a high probability of falling into the unmapped space if it has no knowledge about the current address. The access to the unmapped space will trigger a segment fault, which generates the signal *SIGSEGV*. In addition to accessing unmapped memory, an attacker may also trigger a segment fault due to a permission exception.

The probing to the mapped memory may cause execution errors because of the unknown memory layout. For example, an arbitrary jump can transfer control flow to an instruction's operand instead of the opcode. A non-existent assembly instruction or an illegal instruction will inevitably cause an execution error, which generates the signal *SIGILL*.

Whether an attacker attempts to find the mapped memory in a huge address space, or search for the available gadgets in the memory of unknown layout, he needs to repeatedly manipulate memory (such as read or execute). An exception will be thrown for any probing failure. As a result, attack Vectors 2, 3, 4, 5, 6 will trigger the signal *SIGSEGV* or *SIGILL* with a high probability due to the ASLR, especially the fine-grained ASLR. Normally, if a process triggers the signal *SIGSEGV* or *SIGILL*, the OS will simply kill it. However, the OS allows the process to handle the signals *SIGSEGV* and *SIGILL* by itself to avoid process crash. This provides favorable conditions for attackers to repeatedly probe the code space.

Vector 1 and Vector 7 do not generate any signals, while they also have their own unique characteristics. Vector 1 needs to frequently allocate memory for the execution entity, and constantly adjust the size of the allocated memory. Vector 7 will execute the code blocks n^*512 GB, n^*8 GB, n^*2 MB, and n^*4 kB away from the target instruction, respectively.

These characteristics mentioned above can be used by MagBox to perceive the attacker's memory probing activities. Therefore, the problem of how to perceive memory probing turns into how to collect and identify these behavior characteristics of execution entities.

The second challenge faced by MagBox is how to identify and track risk functions. Risk functions can provide attackers with available gadgets, which are potential candidates for CRAs. In fact, a risk function contains at least a piece of arbitrary jump code, which can be in the form of *ret*, *jmp/call *register*, *jmp/call *pointer*, etc. The purpose of attackers' probing is to find the gadgets that contain arbitrary jump code. More seriously, some risk functions can be used directly without probing. The attacker can deploy the attack by designing the input of the risk function according to the known vulnerabilities. Fortunately, such a single function cannot be used to build a complete gadget chain containing multiple gadgets. However, the risk function itself can also probe memory to search for the available gadgets. For example, BROP [7] can constantly tamper with return addresses based on stack overflow, which can find more available gadgets. In theory, the risk function can not only provide gadgets for CRAs, but also be used as a probing tool by attackers to find more gadgets, which is very dangerous. The risk functions can be identified by perceiving and analyzing memory probing activities.

In fact, not all memory probing activities are triggered by attackers. Some misoperations can also cause the illusion of memory probing. For example, unreasonable memory references (such as wild pointers) may cause an arbitrary read that generates a signal *SIGSEGV*. Furthermore, even the risk functions are not always malicious. Risk functions that are picked and used by attackers will exhibit malicious properties during execution. These malicious properties are contained in the control flow transfers of the risk functions. Therefore, the control flow transfers of the risk functions need to be tracked and collected to judge their legitimacy.

The third challenge faced by MagBox is how to judge whether a risk function is called legally. A risk function behaves maliciously if and only if its control flow is hijacked by an attacker. Afterwards, it can probe memory through an arbitrary jump, or be connected into a gadget chain.

The differences between legal calls and illegal calls will show up in the control flow paths of risk functions. For example, the code *jmp *rax* (a gadget) in the risk function will jump directly to the library function without going through the PLT, which is obviously illegal. However, not all control flow paths' legitimacy can be judged through a single jump. So, we need to combine the history paths and the current instruction to judge whether the risk function is called legally.

Around the three challenges mentioned above, the overall architecture of MagBox is designed as a multi-module coupled structure, as shown in Fig. 1. First, MagBox builds a resource access control module. This module is based on EPT (Extended Page Table) and VMX (Virtual Machine Extensions). It can monitor the running state of the execution entities in real time. Moreover, it can also capture specific events in the OS, such as the process creation and specific instructions' execution.

Next, a fake space mechanism is built. When a process is created, we allocate a large-scale fake space for it. We use memory virtualization to map the fake space to a physical page (*fake page* in Fig. 1) that is unreadable, unwritable and inexecutable. So, a large part of unmapped space become mapped space. Therefore, the new problem faced by attackers is how to filter out the real code segment from a huge fake space. In the original probing case, the access outside the mapped space only triggers *SIGSEGV*, which does not hinder the subsequent probing if attackers can handle the signal by themselves. When the fake space is enabled, the access outside the mapped space will fall into the fake space with a high probability. The probability is determined by the size of the fake space, and we will describe it in *Section* 5.2. Any access to fake space will trigger an access exception (①), which will be perceived by MagBox. Through exception analysis, we can identify the risk functions.

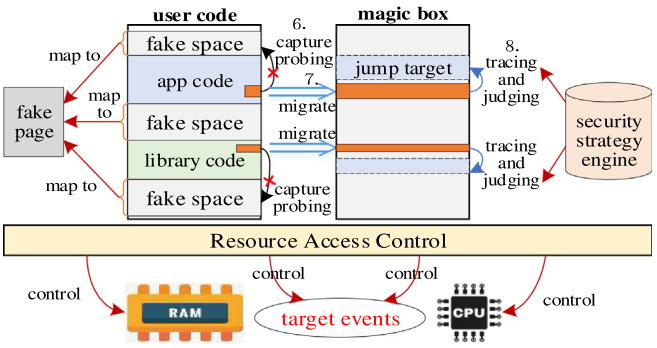


Fig. 1. The overall design of MagBox.

However, memory probing may also access areas outside the fake space, including mapped and unmapped areas, which cannot be captured by MagBox. The mapped area includes executable code segments and non-executable data segments. *SIGSEGV* will be triggered if the memory access is in an unmapped area, or the control flow is transferred to the data segment. If control flow is arbitrarily transferred to a code segment, it may execute an illegal instruction, causing the signal *SIGILL*. To capture these signals, the fake space mechanism also monitors the system call *signal*. Therefore, if an attacker attempts to handle *SIGSEGV* and *SIGILL* by himself, his probing intention will be perceived. The function that can trigger *SIGSEGV* or *SIGILL* will be treated as a risk function, which may be picked by attackers. If the attacker does not handle the two signals, the OS will kill the current process, preventing it from further probing.

After the risk function is identified, its control flow needs to be tracked. If the risk code inside the risk function can be used to illegally transfer the control flow, it means that the control data on which the risk code relies can be tampered with by attackers. As a result, all control flow transfers of this function are untrusted. For example, if an attacker can exploit a stack overflow vulnerability to tamper with a certain return address of a risk function, it can also tamper with other control data (such as a function pointer) that has been loaded on the stack by controlling the number of bytes to be overwritten. Therefore, all the control flow transfers of the risk function need to be tracked to judge their legitimacy.

To track the control flow transfers of the risk function, the risk function will be migrated to a magic box (②). Magic box is a new set of process address space. For example, the native code space of the process is *0x400000-0x480000*, and the new code space in the magic box may be *0x5fd0700000-0x5fd0780000*. The binary code of the risk function will be copied to the magic box. Initially, there is only the risk function in magic box, and all other spaces are redirected to a physical page that is unreadable, unwritable and inexecutable. Any ICT instruction that jumps out of the risk function will trigger an exception, which will be captured by MagBox.

The control flow transfers of risk functions are not always malicious, but they are potential targets for attackers. Therefore, a security strategy engine is needed to determine their legitimacy. The security strategy engine routes legitimate control flow to the appropriate location and prevents illegal control flow transfers (③).

5. The implementation of MagBox

MagBox has 3 basic tasks: identifying the risk function, tracking the control flow of the risk function, and judging the legitimacy of the control flow transfers. In this chapter, we introduce the implementation of MagBox in detail around the 3 tasks.

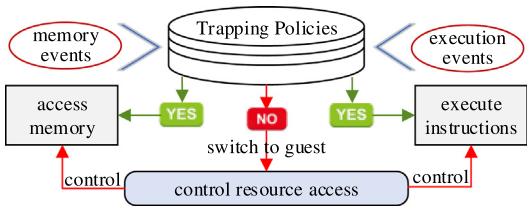


Fig. 2. Resource access control mechanism.

5.1. Build resource access control mechanism

To accomplish the three tasks mentioned above, the processes' access to resources, such as CPU and memory, needs to be monitored and controlled. The native OS does not provide users with resource access control interfaces. MagBox uses virtualization technology to control processes' access to resources, which is shown in Fig. 2.

This mechanism uses the VMX Non-Root and VMX Root to divide the native OS into two modes: *guest* and *host* [43]. Under normal cases, the memory access and instruction execution are performed in the *guest*. When a specific event occurs, the OS will fall from the *guest* mode to the *host* mode (called system trap in this paper). The events that trigger system traps include memory events and execution events, which are set by trapping policies. After the system trap occurs, the current process will be suspended, and the control flow will be taken over by MagBox. In *host*, we can detect the running state of the current process, manipulate the resource to be accessed, set specific trapping events, and determine the direction of the subsequent control flow. In short, the resource access control mechanism provides the basic operating conditions to monitor and control the behaviors of execution entities.

5.1.1. Set memory events

The resource access control mechanism uses EPT to manage all physical memory. It can manage the permissions (read, write, and execution) of the memory by modifying the last three bits of the entry in the last level page tables of EPT. Any memory access that violates permission settings will trigger a system trap. This method can manage the memory permissions with page granularity, but every adjustment requires the OS to fall into the *host*. In contrast, the instruction *vmfunc* can switch the entire set of EPT without triggering a system trap. By modifying the entire set of EPT, we can adjust the memory permissions in a wide range.

In addition to managing memory permissions, this mechanism can also manage the space range of the memory allocated to processes. It uses the kernel function *do_mmap* in the *host* to expand the process's virtual address space. By modifying the addressing page tables of the newly added virtual addresses, we can redirect the memory access to a specific physical page. In addition, by rewriting the entries in the last-level page table of the EPT, we can also achieve physical page redirection.

5.1.2. Set execution events

Event control includes two steps: event injection and event capture. The corresponding fields (such as *vm-execution*) in VMCS (Virtual Machine Control Structure) will be modified to inject new events into the OS. These events can change the execution entities' running state, target resource, and control flow paths. When a specific event occurs, the OS will fall into the *host*. Then, we know which specific event occurs. The controlled events include breakpoint setting, general protection exception injection, specific instruction capturing and context rewriting.

Breakpoints can be divided into data breakpoints and instruction breakpoints. When setting a breakpoint, the instruction address should be written into the debugging registers (*Dr0*∼*Dr3*). Then the *R/W* bits corresponding to the debugging register in *Dr7* is set to enable it a writing breakpoint, reading breakpoint, or executing breakpoint. When the OS operates the breakpoint in *guest* mode, it will trigger a system trap. By setting breakpoints, we can control the execution entity's resource access and instruction execution with byte granularity.

The malicious activity can be stopped by injecting a general protection exception into the OS. When an activity needs to be stopped, MagBox sets the bit *31* of the *vm-entry interruption* in VMCS to 1. Then it sets bits *10:8* to *011* (*hardware exception*). Finally, bits *7:0* are set to *00001011* (interrupt *13 #GP*). After that, the OS will generate a *#GP* when running in the *guest* again, which prevents the current operation.

Some fields in VMCS provide basic conditions to capture the execution of specific instructions (such as *int3, mov to cr3* and *hlt*, etc.). The execution of these instructions will cause system traps.

When the OS falls into *host*, we can reset the CPU context by rewriting the *guest fields* in VMCS. For example, the control flow redirection can be realized by rewriting the *guest rip* in VMCS. The new CPU context determines the execution entity's control flow path and target resource, which provides conditions for controlling the OS's behavior.

5.2. Perceive memory probing and identify risk functions

Due to ASLR (especially the fine-grained ASLR), attackers cannot know the accurate address of the target function before code probing. The purpose of code probing is to search for the code with potential attack capabilities (gadget or dispatcher-gadget), or crack the randomized address. If the code probing can be detected, on the one hand, we can detect the probing intention of the attacker; on the other hand, we can locate the potential gadget (i.e., arbitrary jump code) that may be selected by the attacker. For example, if an attacker uses the arbitrary jump code to probe available gadgets, the arbitrary jump code itself is a gadget or a dispatcher-gadget [44]. The idea of MagBox is to use the attacker's code probing ability to identify the risk functions containing gadgets.

The fake space mechanism is designed to perceive the memory probing, as shown in Fig. 3. When a target process is created, we use the kernel function *do_mmap* to allocate a huge fake space (about 30 TB in this paper) for it. The fake space will be redirected to an unreadable, unwritable and inexecutable physical page (*fake code* in Fig. 3) by rewriting the corresponding page table entries. It should be noted that there is no relationship between the permission set by *do_mmap* and the permission set by EPT. For example, a memory area can be mapped as readable and executable, but it may be set as unreadable and inexecutable by EPT, which is transparent to attackers.

In addition, fake space mechanism modifies the system call *signal* to capture the activity that the current process handles the signals *SIGSEGV* and *SIGILL*. The *signal* in the system call table will be redirected to a new code block that can detect if there is a signal *SIGSEGV* or *SIGILL* to be handled. If yes, it executes instruction *int3* to trigger a *system trap*. After that, MagBox records the instruction triggering the signal. The risk function can be located by analyzing the instruction.

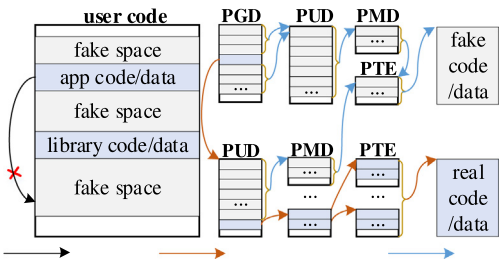


Fig. 3. Fake space mechanism. Gray: represent the virtual fake space, the fake space addressing items, and the fake code that are unreadable, unwritable, and inexecutable. Blue: represent the user's native virtual address space, the addressing table entries of the native space, and the real binary code that is executable.

Among the currently known probing technologies, allocation oracle, arbitrary write, and process cloning (i.e., Vectors 1, 2, 3) cannot accurately locate the gadgets that are known in specific forms, nor can they filter out the gadgets that are unknown but available. They are often used as aids in conjunction with other probing techniques. Therefore, we cannot identify risk functions by analyzing the activities of Vectors 1, 2, 3. But they still expose the attacker's probing intentions, and will be stopped by MagBox.

Vector 1. Under the protection of the fake space, the memory areas probed by allocation oracle (Vector 1) cover all fake spaces. The native code area of the process occupies only a small part of all the mapped areas. Attackers cannot identify which segment is the real code from the mapped areas. This makes the allocation oracle lose the probing meaning.

Vector 2. Arbitrary write (Vector 2) can trigger an arbitrary jump by tampering with control data, which will be discussed later. It can also probe the attributes of the memory according to the crash information. If the mapped space has been known (provided by Vector 1), writing data to different addresses in the space will cause different results. If the data can be written into the target address without any exception, the probed target is a writable data area. If the signal *SIGSEGV* is triggered, the probed target is in an unwritable data area or a code segment. This information can help attackers to further reduce the memory range to be probed. In fact, Vector 2 is difficult to bypass the fake space mechanism. When it probes code segment, it either triggers *SIGSEGV* or triggers an EPT violation, which will be captured by MagBox.

Vector 3. The address space of the child process is the same as the address space of the parent process, and the crash of the child process does not affect its parent process. Vector 3 can use this feature to frequently probe memory layout without leading to parent process crash.

MagBox detects process cloning by checking the system call *fork*. The cloned process contains a same address space (including the fake address space) with parent process. MagBox sets the address space of child process to the exact same permission configuration as the address space of parent process. Therefore, the memory probing to the child process can also be perceived, which makes the Vector 3 unable to bypass MagBox.

Vector 4. Unlike the above 3 Vectors, arbitrary read (Vector 4) can read the code and directly probe the available gadgets. In the process address space, the code segment, data segment, heap and stack are all readable. Therefore, Vector 4 can read a lot content from memory. For example, *heart bleed* can read 64 kB memory in one probing.

The contents that processes can read are limited to the data area. In the known attack scenarios, the attacker needs to cross the data area to read the code segment. For example, if HeartBleed wants to read the process code, it needs to continuously reduce the *pl* in the function *memcpy (bp, pl, payload)* until the *pl* points to the code segment.

The fake space mechanism sets the adjacent areas of each code segment to be unreadable. The size of the unreadable area is far larger than the size of the code segment. Unless the attacker can know the specific code segment range, it will read an unreadable area at a high probability, which can be captured by MagBox. For example, if the code space of the process is *30* MB and the fake space is *30* TB, the attacker's single probing success rate is only *1/1000000*. Moreover, attackers need to repeatedly probe memory to get available gadgets, and any failure will be captured. Therefore, under the protection of MagBox, Vector 4 cannot read meaningful code, and cannot probe the risk function containing gadgets.

Although the attacker can read some control data through Vector 4, the small amount of control data does not provide enough gadgets to the attacker. In ASLR with function as granularity, a single piece of control data can only reveal the address of a single function.

Vector 5. DOP [42] exploits data-leak (Vector 5) to read the relative address stored in PLT to obtain the GOT address, where stores all the library function addresses called by the current process. Compared with other memory leakage, Vector 5 can bypass fine-grained ASLR and get more addresses through simple calculation units.

To detect Vector 5, the PLT will be set to be unreadable (set by EPT). In normal scenarios, the process does not read its own code, let alone read the library code it relies on. Therefore, the setting does not affect the normal execution of the process. When the attacker reads PLT, his activity will be captured by MagBox.

It should be noted that we do not set all the code to be unreadable, which is expensive. In Linux, the code is loaded to the memory when it is executed for the first time. Therefore, we cannot set all the code to be unreadable at one time, which may miss the attack process. Although we can track the code page allocation by setting the present tag (*p*) or hooking the page fault handler, this will seriously reduce the running speed of the process. We only set the PLT to be unreadable, and the code reading probing is handled by fake space mechanism mentioned above.

Vector 6. For arbitrary jump (Vector 6), when the control flow jumps to fake space, the OS will fall into *host*. After that, we can get the address of the arbitrary jump code through the LBR registers. It should be noted that we do not need to worry the attacker will confuse the information stored in the LBR. Because when an abnormal instruction occurs, we only check the top pair of the LBR registers, not all of them. If the attacker still uses the method obfuscating LBR to prevent us from analyzing his behavior, then he will expose himself faster. The reason is that frequent jumps will increase the probability of jumping to the fake space, which increases the risk of illegal control flow transfers being detected.

When the control flow jumps to a mapped but inexecutable area, *SIGSEGV* will be triggered. When the control flow jumps to illegal instructions, *SIGILL* will be triggered. Under fine-grained ASLR, it is impossible to jump to the available gadgets every time, even he knows which area is the real code segment. The attacker will inevitably trigger *SIGSEGV* or *SIGILL* in the multiple probing activities. For example, we implement BROP by modifying the return address to a 64-bit random value, which triggers *SIGSEGV* at a probability of more than 99%. If only the last 12 bits of the return address are modified to a random value, the probability of triggering *SIGILL* is than 90%; If such operations are continuously performed 3 times, the probability of triggering *SIGILL* is more than 99%. Therefore, almost all the arbitrary jump cannot avoid MagBox's detection.

The code block containing ICT instructions can transfer the control flow to a non-fixed position. If it transfers the control flow into the fake space or triggers the signals *SIGSEGV* or *SIGILL*, it means that the control data can be maliciously tampered with. The function in which such a code block resides is a risk function.

Vector 7. Vector 7 exploits the TLB hit information to crack the 12th to 47th bits of the randomized address. In the process of converting a 64-bit 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 4 kB, 2 MB, 1 GB, and 512 GB memory, respectively. To obtain the last 3 bits of the index value of the page table at each level, Vector 7 needs to access the memory which is separated from the virtual address by n*4 kB, n*2 MB, n*1 GB, and n*512 GB respectively (n $<$ 8). For example, to get the index value of the virtual address V in the third-level page table, Vector 7 needs to execute $V + n^*1$ GB (n $<$ 8) in sequence until the current cache line is filled.

The areas away from the code segment n^*1 GB and n^*512 GB (n $<$ 8) are mapped as unreadable, unwritable, and inexecutable, if these areas have not been mapped. When Vector 7 visits these areas, it will be captured by MagBox. As a result, an attacker cannot crack the bits 30th∼32nd and 39th∼41st of the virtual address. However, before the current probing, Vector 7 has already cracked the address bits15th∼20th, 24th∼29th, 33rd∼38th, and 42nd∼47th. Although we can prevent the current address from being completely cracked, the information entropy of the code address being probed has become very limited, which can greatly reduce the difficulty of the attack. When perceiving Vector 7, MagBox will determine the function where the probed code is located as a risk function.

In summary, when the Vectors 1∼5 is detected, attackers cannot obtain available gadgets or crack the randomized address. Therefore, the probed function does not provide them with meaningful code or addresses. In contrast, Vector 6 itself is an available gadget. Such a vector even can be directly used as a gadget or a probing tool without probing. For example, BROP only needs to know the relative offset between the array boundary and the return address, and it can use the stack overflow vulnerability to tamper with the return address to perform code probing. For Vector 7, when it is detected, the attacker has cracked most bits of the target address. Therefore, the functions containing Vector 6 or Vector 7 has the risk of being used maliciously, and they will be treated as risk functions.

5.3. Track control flow

A risk function contains at least one code block with potential attack capabilities, which can transfer control flow to the next gadget. Therefore, the control flow transfers of the risk function need to be tracked to determine their legitimacy.

In the native address space, risk functions are mixed with other code. To track all the control flow that jumps out of the risk function, the traditional method directly modifies the jump branches of the risk function. However, this method is usually compiler-based. It needs to identify risk functions before the code is compiled. This contradicts the original intention of dynamically identifying risk functions. Control flow tracking can also be achieved by changing the code permissions in the native space. For example, when the risk function is executed, we set all other code as inexecutable. So, when the control flow jumps out of the risk function, it will be captured. However, the granularity of permission adjustment is pages, and the risk function does not occupy an integer number of pages. Therefore, directly adjusting the permissions of the risk function will affect the execution of other code.

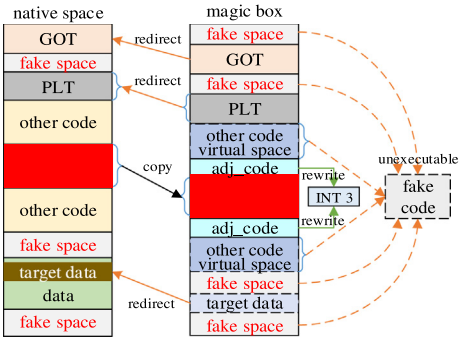


Fig. 4. The risk function migration.

To track the control flow transfers of the risk function, we migrate the risk function to a magic box with specific space structure and permission configuration, as shown in Fig. 4. The virtual space size of the magic box is the same as the native space of the process, and they do not overlap. The page(s) where the risk function is located will be completely copied to the magic box. In the magic box, except for the binary code of the risk function, all other copied binary code (*adj_code*) will be rewritten to *0xcc* (*int3*). At the same time, we set the instruction *int3* as a system trap event. Another word, the magic box only contains the risk function's code and the modified code *0xcc*, and the rest of the space is fake space that is unreadable, unwritable, and inexecutable.

It should be noted that we do not migrate the PLT, GOT and data segments on which the risk function relies to the magic box. When the risk function accesses the PLT, GOT or data segments, a system trap will be triggers. After that, we redirect the memory access in the magic box to the real physical page(s) corresponding to the PLT, GOT or data segment. This design can limit the resources that can be accessed by risk functions and ensure normal data access.

The control flow paths of the risk function are shown in Fig. 5, and the control flow transfer method is shown as Algorithm 1. When a risk function is migrated to the magic box, the head of the risk function in the native space will be rewritten as *jmp $ept_switch* (①). The code in *ept_switch* uses the instruction *vmfunc* to switch the current EPT to a new set of EPT. In the new EPT, the code in native space is inexecutable, while the risk function in the magic box is executable. To prevent the attacker from directly jumping to the ICT instructions of the risk function in the native space, the ICT instructions will be marked with *int3*. In addition, the EPT switching code in *ept_switch* and *ept_back* will be set to be unreadable and unwritable to prevent the EPT index *v_0* and *v_1* from leaking. At the same time, *v_0* and *v_1* have been randomized to prevent fake *vmfunc* attacks. Moreover, it can also prevent the magic box address from being leaked.

Algorithm 1: The control flow (CF) transfer method

Input: The CF that jumps to the risk function (RF) header in native space
Output: The CF that returns the RF in native space

```
1. Rewrite(RF, jmp $ept_switch) // rewrite the RF's header
2. Foreach  RF header← CF  do
3.     Transfer(CF, ept_switch) // transfer CF to ept_switch
4.         Foreach ept_switch ← CF do
5.             Switch(EPT_1, EPT_2) // switch EPT
6.             RetAddr = DumpRetAddr(rbp) // record return address
7.             PushAddr(ept_back) // store the return address on stack
8.     Transfer(CF, MagicBox) // transfer CF to Magic Box
9.         Foreach MagicBox ← CF do
10.          If ept_back ← CF then
11.             Switch(EPT_2, EPT_1) // switch EPT
12.             CheckRetAddr(RetAddr, rbp) //check the return address
13.    Transfer(CF, NativeSpace) // transfer CF to native space
14.        NativeSpace ← CF
```

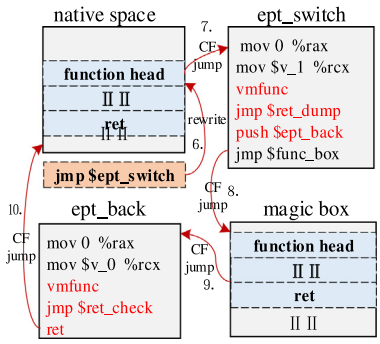


Fig. 5. Control flow transfers between the native space and the magic box. Blue spaces: risk functions in two spaces. Red instruction: The instructions affecting control flow transfers.

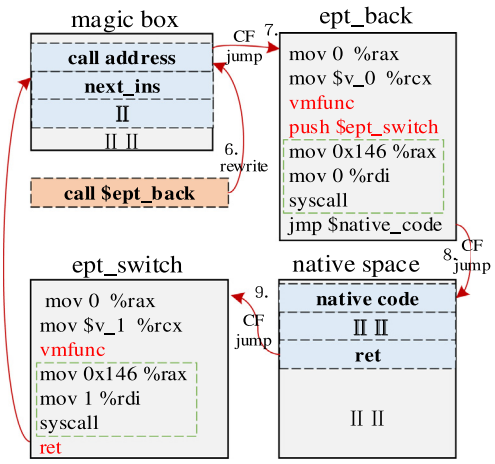


Fig. 6. Legal control flow redirection. Blue: risk functions in two spaces. Green: protect the return address.

When the risk function in the native space is called again, the control flow will jump to *ept_switch* for EPT switching (②). After that, the stack will be adjusted to protect the return address, which will be described in the next section. The address of *ept_back* whose code can restore back the original EPT will be pushed onto the stack. Next, the instruction *jmp $func_box* redirects the control flow to the risk function header in the magic box (③). At this point, the risk function starts to be executed. When the risk function returns to its caller, the instruction *ret* will use the address of *ept_back* previously pushed onto the stack to redirect control flow to *ept_back* (④). After the original EPT is restored, the instruction *ret* is called again. Then, the stack is recovered, and *ret* uses the real return address of the risk function in the native space (⑤). Finally, control flow returns to the native space again. In MagBox, unless the control flow jumps out of the risk function, it will not trigger any system trap.

In the magic box, any control flow that jumps out of the risk function will trigger a system trap because of *int3* or permission exception. The legal control flow will be redirected to the native space, and the illegal control flow will be stopped by injecting a general protection exception into the OS. How to determine the control flow's legitimacy will be introduced in the next section.

5.4. Judge the legitimacy of control flow

CRAs redirect control flow to the selected gadgets by tampering with the control data associated with arbitrary jump code (ICT instructions). Not all control flow transfer instructions can be used as gadgets. Only those instructions whose jump targets pointed by the writable control data can be used as gadgets or dispatcher-gadgets. Other instructions will be judged to be legal.

The illegal control flow transfers between gadgets have their unique characteristics. These characteristics are key to developing security strategies. The security strategies adopted by MagBox are as follows:

(1) The instruction *call address* is legal when it transfers control flow outside the risk function.
(2) The return address cannot be changed before the *ret* is executed.
(3) *jmp* * only transfers control flow 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.
(4) 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.
(5) The jump targets of ICT instructions must conform to the code alignment forms in the ELF file.
(6) For the control data, if it originally points to the head of the function, it will not point to the inside of the function after being overwritten; if it originally points to the inside of the function, it will not point to the head of the function after being overwritten.
(7) Control data does not become non-control data after being updated, and non-control data does not become control data after being updated.
(8) For the control data in the form of local variable, it cannot be modified by the code that can write non-fixed number of bytes into memory after it has been assigned but not read.

5.4.1. Transfer the legal control flow

For the legal control flow transfer instruction *call address*, we do not need to track its control flow. It will be directly redirected to the native space. The redirection path is shown in Fig. 6, and the control flow transfer method is shown as algorithm 2.

Algorithm 2: The legal control flow (CF) redirection

Input: The CF transfer caused by *call address* in magic box
Output: The CF that returns magic box
1. *Rewrite(RF, call $ept_back)* // rewrite the *call address* in RF
2. **Foreach** *call address* ∈ *MagicBox* **do**
3. *Transfer(CF, ept_back)* // transfer CF to ept_back
4. **Foreach** *ept_back* ← *CF* **do**
5. *Switch(EPT_2, EPT_1)* // switch EPT
6. *PushAddr(ept_switch)* // store the return address on stack
7. *Syscall(0x146, 0)* // set a breakpoint to protect return address
8. *Transfer(CF, NativeSpace)* // transfer CF to native space
9. **Foreach** *NativeSpace*← *CF* **do**
10. **If** *ept_switch* ← *CF* **then**
11. *Switch(EPT_1, EPT_2)* // switch EPT
12. *Syscall(0x146, 1)* // cancel the breakpoint
13. *Transfer(CF, MagicBox)* // transfer CF to Magic Box
14. *MagicBox*← *CF*

The instruction *call address* in magic box is rewritten as *jmp ept_back* (①). When the instruction is executed, the control flow will be redirected to *ept_back* (②). After the EPT switch is completed and the address of *ept_switch* is pushed onto the stack, the control flow will directly jump back to the native space (③). After the execution ends in the native space, control flow will jump to *ept_switch* through the instruction *ret*, because the address of the *ept_switch* has been pushed onto the stack (④). After the EPT

switch is completed, the control flow returns to the magic box again (⑤). For the risk function migrated to the magic box, we locate all instructions *call address* and redirect their control flow to *ept_back*. As a result, these instructions do not trigger system traps when executed, which reduces performance overhead.

5.4.2. Protect the backward control flow

There are two types of return addresses to be protected. One is the return address generated when other functions call the risk function. The other is the return address generated when the risk function calls other functions. For the former, we record the return address when other functions call the risk function, which is shown as the *jmp $ret_dump* in Fig. 5. Before the *ret* in the risk function is executed, we check whether the return address stored on the stack is the same with the recoded one, which is shown as the *jmp $ret_check* in Fig. 5.

In fact, after the risk function is migrated to the magic box, the control data on the stack includes the *ept_back* address and the original return address. Both can be tampered with by the stack overflow. However, this does not affect the security of MagBox. First, the code in the native space is inexecutable. Even if an attacker can tamper with *ept_back* address, he cannot execute the code in the native space. The reason is if without EPT switch, the code is inexecutable. Second, if the attacker executes *ept_back*, then he will be detected due to the changed return address.

In the risk function, the return address is generated by the instructions *call address* and the *call *xx*. *call *xx* will trigger a system trap due to transferring the control flow to the native code space. After that, MagBox sets a writing break point to the position which stores the return address on the stack. When the legal control flow returns, the breakpoint will be canceled. When attackers tamper with the return address, the breakpoint will be triggered, which will be captured by MagBox.

Compared with *call *xx*, *call address* does not trigger a system trap. It runs in ring3. Therefore, we cannot directly protect the return address by setting breakpoints, otherwise it will cause execution errors. To solve this problem, we added a new system call *ret_pro* to OS. Its system call number is 326 (0×146), and it has a parameter *flag*. When *call address* is called in magic box, the control flow will be redirected to *ept_back*. Then, the 346th system call will be executed and the OS enters the ring0. If the flag is 0, the 16 bytes (the return address and the *ept_switch* address) pointed by the register *rsp* are set to be unwritable by the register *dr2* and *dr3*, which is shown as the green dotted frame in Fig. 6. If *flag* is 1, *ret_pro* will cancel the breakpoints set by *dr2* and *dr3*. As a result, attackers cannot tamper with the return address and *ept_back* address stored on the stack.

5.4.3. Protect the forward control flow

The security strategies 3∼8 are used to protect the forward control flow transfers. When ICT instructions (*call *xx* and *jmp *xx*) in magic box transfers control flow to the native space, a system trap will be triggered. After that, we exploit these security strategies to judge the legitimacy of the forward control flow transfers. Security strategies 3∼5 can be used directly when a system trap is triggered by ICT instructions. In contrast, the strategies 6∼8 need to locate the control data of ICT instructions.

When *call *xx* or *jmp *xx* is captured due to the system trap, we first analyze the executed code blocks recoded by LBR to find the operations related with the control data, including data reading and data writing. However, LBR can only record 16 code blocks, and it may not be able to record all the operations related with control data. If LBR is not enough, we will enable Intel PT when the risk function is called again to capture more code blocks. The method locating the control data of ICT instructions is shown as Fig. 7. In fact, even multiple memory dereferences can be

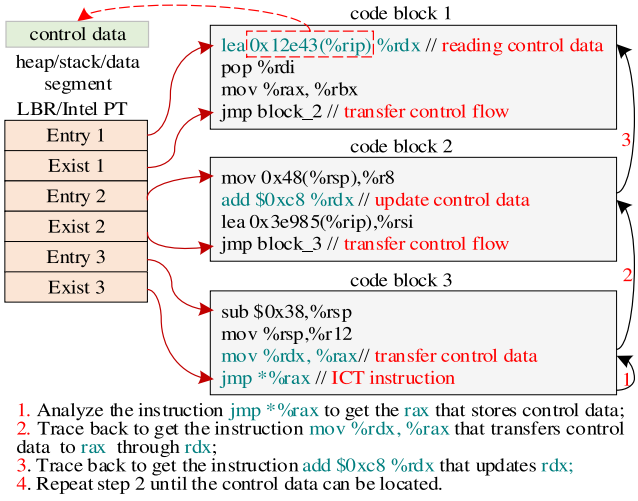


Fig. 7. Locate the control data.

```
905:  48 8b 45 a8      mov   -0x58(%rbp),%rax // pass a pointer on stack to rax
909:  48 8b 40 10      mov   0x10(%rax),%rax// pass the value in the memory
pointed to by pointer+0x10 on the stack to rax
90d:  bf 1e 00 00 00   mov   $0x1e,%edi // setting a parameter
912:  ff d0            callq *%rax // call the function pointed by *(pointer+0x10)
```

Fig. 8. Locate multiple memory dereferences.

detected. Because during the memory dereferencing process, the code iteratively reads the data in the memory into the registers until the end of the dereferencing, as shown in Fig. 8.

The objects storing control data include heap, stack and data segment. The types of control data include local variables and global variables. The former is stored in the stack or heap, and the latter is stored in the heap or data segment. If the control data is a global variable, it may have been tampered with before the risk function is called. As a result, we cannot detect the changes of the control data during the running of the risk function, which makes strategies 6∼7 unable to be applied.

Fortunately, under fine-grained ASLR, the attacker does not know the code address until the code is probed. That is, the multiple code probing is inevitable. When an attacker repeatedly tampers with the same global variable (control data) to find available gadgets, we can collect all the changed values of the control data. After that, the strategies 6∼7 can be used to judge the legitimacy of the control flow transfers.

For the control data in the form of local variable, it is not defined and assigned until the risk function is executed. It becomes invalid after the risk function exits. Therefore, we only need to observe the changes in the target control data during the risk function running. In addition to complying with security strategies 3∼7, the control data must also comply with security strategy 8.

If the control data in the form of local variable is stored in the heap, we need to locate the heap allocation (*malloc* or *new*), and calculate the offset between the location of the control data and the first address of the heap. If the control data is stored on the stack, we need to calculate the offset between the location of the control data and the start address of the current function stack frame. Then, we can get the location of the control data.

To tamper with local variables, attackers need to call the functions or code blocks that can overwrite memory. Although *printf (%n)* can accurately modify the target data, it has obvious attack characteristics and is easily detected. It outputs a lot of

Table 1
Library functions adopted by CRAs.

Header	Functions
⟨string.h⟩	strcpy(), strncpy(), strccpy(), strcat(), strdup(), memcpy(), bcpy(), getchar()
⟨stdio.h⟩	scanf(), sprintf(), snprintf(), fprintf(), vsprintf(), sscanf(), fscanf(), gets(), fgets(), vfscanf(), vscanf(), vsscanf, getc(), fgetc()
⟨libgen.h⟩	streadd(), strcadd(), strecpy(), strtrns()
⟨stdlib.h⟩	realpath()
⟨conio.h⟩	getch()
App code	rep xx xx xx xx, LOOP code block

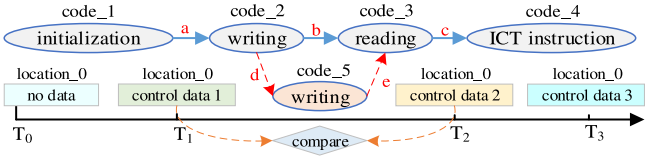


Fig. 9. Track the control data processing.

0 or spaces, or modifies a complete control data byte by byte, which reveal its attack intention. By checking the parameters of *printf*, we can filter out the *printf* with potential attack intention. In addition, the deployment conditions of *printf* attack are very strict. The attacker needs to precisely control several parameters, which can only be done through external input, such as (*scanf("%s", buffer)*; *printf(buffer)*). The attacker also needs to accurately calculate the offset between the target data and the input, which is almost impossible under the ASLR protection. Therefore, *printf (%n)* is hardly adopted by CRAs.

Compared with *printf (%n)*, the functions that can continuously write non-fixed number of bytes into memory are more likely to be adopted by CRAs, which are shown in Table 1. These functions have a common feature that they can cause memory overflow under specific conditions. Although some functions (such as *getc()*) cannot directly cause overflow, they are possible in a loop structure. For example, in a looping code block, the number of calling *getchar()* is controlled by the input, and the code block can write a non-fixed-size string to the stack, which may also cause a stack overflow. Moreover, all the instructions *rep xx xx xx* and LOOP code blocks that can write multiple bytes to memory can also be adopted by attackers. For example, if *%es:(%rdi)* in the instruction *rep stos %rax, %es:(%rdi)* points to the stack, and the number of bytes to be written is determined by *rcx*, this code block can cause stack overflow when *rcx* can be controlled.

For the control data in the form of local variable, attackers must tamper with it after it has been assigned. Otherwise, the tampered data will be restored to a legal value by the legal assignment statement. Different from the legal control data processing, the illegal one will add an additional writing operation between the data writing and data reading, which is done by the functions or code blocks mentioned above. If we can find this additional writing operation, we can detect the illegal control flow transfer.

We set a breakpoint to capture data reading and data writing at the location of the control data. When the risk function is called, the operations (reading and writing) on the location of the control data will trigger system traps. Those assignment statements (or functions) and memory reading statements (or functions) are recorded. Combined with the code blocks collected by the LBR/Intel PT, we can screen out the assignment statement and memory reading statement related to the control flow transfer, as shown in Fig. 9.

In the magic box, the code blocks containing assignment statements (such as *mov %rax, 0x279ee7(%rip)* and *call memcpy@plt*) and the code blocks containing memory reading statements (such as *mov 0x277092(%rip), %rdi* and *strcpy@plt*) will be redirected to two different functions, respectively. The two functions can record the control data that has been written into memory and the control data that has been read, respectively. If the two pieces of data are the same, it means that there is no additional control data modification between the assignment statement and the memory reading statement. Otherwise, there must be an additional write operation between the write operation and the read operation, which will be reflected by setting a *com_flag*. It should be noted that tracking control data does not trigger any system traps.

In Fig. 9, *code_2* and *code_3* determine where *code_4* will transfer the control flow. The propagation path of the control data is $a \rightarrow b \rightarrow c$, which is provided by LBR/Intel PT. If a *code_5* that can rewrite the control data is added between *code_2* and *code_3*, then *code_2* will no longer be able to determine the control flow transfer of *code_4*. At the same time, the propagation path of the control data becomes $a \rightarrow d \rightarrow e \rightarrow c$. At this point, *com_flag* is 1.

In practice, even if *com_flag* is 1, it does not mean that the current control flow transfer is illegal. To judge the legitimacy of the control flow transfer, we need to check 3 things when there is a system trap caused by an ICT instruction. The first thing is to check whether the code that transfers control flow is *code_4*. If the control flow transfer code is not *code_4*, it means that the risk function contains another ICT instruction. The newly discovered ICT instruction is a new detection target that has its own control data propagation paths. The second thing is to check whether the control data propagation path that can determine *code_4's* control flow transfer has been changed. That is, whether the control data propagation path used by *code_4* has been changed from the original $a \rightarrow b \rightarrow c$ to the current $a \rightarrow d \rightarrow e \rightarrow c$ or some other paths. The third thing is to check whether the new write operation is performed by the function or code block in Table 1. If the above three checking results are all yes, the current control flow transfer will be judged to be illegal.

When the control data is stored in data structures, arrays, or classes, it may be initialized by the functions mentioned in Table 1, which may lead to a misjudgment of strategy 8. However, after the control data is written into memory and before it is read, data update is rarely performed by these functions. To verify this strategy, we analyzed various applications such as *Redis*, *SpecCPU2006*, *Lmbench*, *Apache*, *libc*, *gzip*, *tar*, and *codeblocks, etc* (more than 200,000 code lines), and found no data update violating strategy 8.

6. Evaluation

We conduct all experiments on a Lenovo desktop equipped with an i7 CPU and 32 GB memory. The OS is Ubuntu-16.04 with kernel 4.4.0.

6.1. Security evaluation

To verify the defense effect of MagBox against CRAs, we use ROPgadget [45] to search for available gadgets and gadget chains. After that, we manually trace the transfer paths of the control flow according to the nodes in the gadget chain. Finally, we use the security strategies proposed in this paper to analyze whether

Table 2
Attack defense of MagBox.

App	LOC	Total gadgets	Gadget chain	Effective
400.perlbench	169 909	100 750	5	Yes
401.bzip2	8293	3942	1	Yes
403.gcc	521 038	254 156	29	Yes
416.gamess	466 415	207 929	2	Yes
435.gromacs	108 559	28 496	3	Yes
450.soplex	41 428	23 553	5	Yes
453.povray	155 163	45 722	12	Yes
454.calculix	166 765	53 729	7	Yes
456.hmmer	35 992	13 063	3	Yes
465.tonto	165 470	82 489	3	Yes
470.lbm	1155	487	6	Yes
471.omnetpp	47 903	56 954	2	Yes
481.wrf	214 948	75 310	27	Yes
482.sphinx3	25 090	7065	3	Yes

MagBox can detect illegal control flow transfers. The experiment results are shown in Table 2. The results show that MagBox can detect all illegal control flow transfers. Because, these control flow transfer activities violate one of the security strategies 1∼8.

Theoretically, MagBox has the possibility of missing judgment. Suppose an application contains enough memory vulnerabilities, and each vulnerability can trigger an arbitrary jump *call *xx*. At the same time, a memory area (such as Vtable) containing multiple function pointers can be read or rewritten, and these pointers can just build a complete gadget chain. Then, the gadget chain can bypass strategies 1∼8. Fortunately, such harsh conditions hardly exist in real applications.

To verify MagBox's defense against real code probing attacks and CRAs, we simulate and deploy the following attacks:

Allocation Oracle. We deploy Allocation Oracle attacks in *Nginx 1.6.2*. During the attack, *ngx_create_pool ()* is used to probe the mapped areas. Under MagBox protection, Allocation Oracle can still get the mapped areas. However, all the areas it probes are very large. The real process code segment and libraries are hidden in these huge mapped areas. As a result, Allocation Oracle cannot get any available code address from these areas. Even the memory area where the sensitive data (such as Vtable) stores, if they are hidden in a huge fake space, they are difficult to locate by Allocation Oracle.

Arbitrary Write. Most of Arbitrary Write attacks are initiated through memory vulnerabilities. We deploy Arbitrary Write in *wget-1.19.1* to probe the hidden code segment. The function *skip_short_body ()* passes a controllable parameter to *fd_read()* to overwrite the current function stack frame. Due to the fine-grained ASLR, the specific code address cannot be known. When arbitrarily tampering with the return address, the probability of triggering signal *SIGSEGV* is so high that it is easy to be captured by MagBox.

Process Clone. The process clone itself does not probe memory. It can provide other probing technologies with the same address space as the parent process address space, thereby avoiding directly probing the parent process. We deploy *ImageMagick 7.0.7-16*, and add a *fork ()* to create a child process. In the child process, the stack overflow vulnerability (*CVE-2017-17880*) in *coders/webp.c/WriteWEBPImage* will be triggered to change the return address to a random value. The result shows that the signal *SIGSEGV* or *SIGILL* will be generated, which can be captured by MagBox.

Arbitrary read. We deploy HeartBleed in *openssl-1.0.1c* to simulate arbitrary read. To read the process code, the parameter *pl* of the *memcpy (bp, pl, payload)* in *openssl* will gradually decrease. This operation can extend the leaked data 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. Under the protection of MagBox, HeartBleed will trigger EPT exception when accessing the fake space, and the signal *SIGSEGV* will be triggered when accessing an unmapped area, which will be captured by MagBox.

Data leakage. We use the DOP to simulate data-leakage based on the method proposed by Hu [42] to extract the GOT address stored in PLT. When we tried to copy the address of the function *system* in PLT to the local variable *p, MagBox detects and prevents the current operation. The reason is the PLT is unreadable.

Arbitrary jump. We use BROP [7] to simulate arbitrary jump. We exploit the function *ngx_http_parse_chunked* to trigger the vulnerability *CVE-2013-2028* in *nginx 1.3.9*, which can arbitrarily tamper with the return address. According to our observations, if only the last 12 bits of the return address are tampered with, *SIGILL* or *SIGSEGV* will be triggered when up to 5 code blocks are executed continuously. If the return address is tampered with a random 48-bit address, the probability of triggering *SIGILL*, *SIGSEGV* and EPT exception exceeds 99%. Whether it is *SIGILL*, *SIGSEGV*, or the EPT exception, it will be captured by MagBox.

Side-channel probing. To simulate side-channel probing, we imitate the principle of AnC [9] to construct a C program, which contains a loop structure that allows users to use the function *mmap()* mapping multiple specified memory areas. At the same time, users can access the mapped areas at will. In the experiment, we found that MagBox cannot detect the activities of cracking the address bits15th∼20th, 24th∼29th, 33rd∼38th, and 42nd∼47th. When cracking the 30th∼32nd and 39th∼41st of the virtual address, an EPT exception is triggered, which is captured by MagBox.

In fact, MagBox cannot prevent all code probing. For example, BROP may have done several code probing and got some available gadgets before triggering the signal *SIGILL*. Fortunately, as long as one probing step can be detected during the whole code probing, MagBox can prevent an attacker from building a complete gadget chain. Because eliminating any node in a gadget chain makes the subsequent gadgets unable to be executed. At the same time, the function related to illegal control flow transfer will be regarded as a risk function, which will be migrated to the magic box. Afterwards, the control flow transfers of the risk function are monitored, which prevents the probed code from being used as gadgets again.

However, MagBox does not have a defensive effect against all CRAs. If attackers can obtain available gadgets without code probing, MagBox will lose its detection and defense effects. Because the attackers will not reveal any risk functions to MagBox if there is no code probing. The absence of risk functions means MagBox will not track any control flow, even if the OS is under attack. Fortunately, technologies such as ASLR, memory hiding, and pointer encryption make it difficult for attackers to gain gadgets without code probing. Code probing has gradually become a necessary part of CRAs.

6.2. Performance evaluation

Lmbench. we use Lmbench to test the runtime overhead of the OS introduced by MagBox, as shown in Fig. 10. The results show that the average overhead introduced by MagBox to OS is 2.3%.

SpecCPU2006. We use SpecCPU2006 to test the CPU performance loss caused by MagBox, as shown in Fig. 11. The results indicate that the average overhead introduced by MagBox is 3.4%.

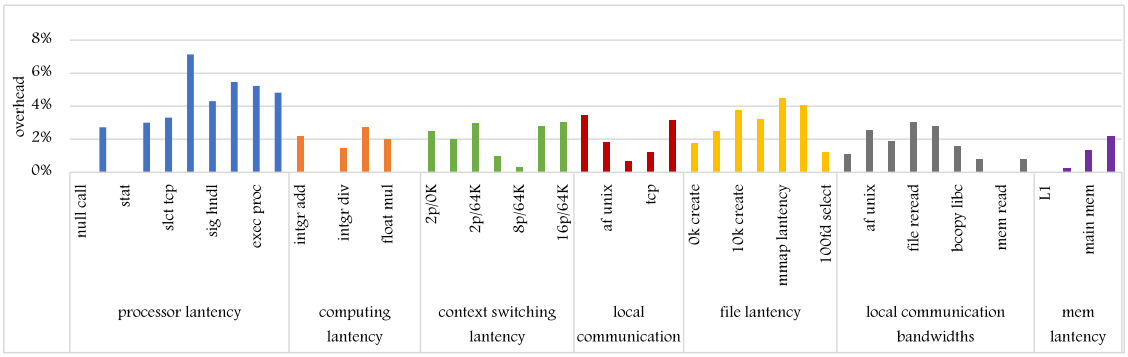


Fig. 10. Lmbench test results. The abscissa indicates the test items, and the ordinate indicates the performance degradation factor. The average attenuation factors of each group (from left to right) are 3.6%, 1.4%, 2.1%, 2.1%, 3%, 1.6 and 0.9%, respectively. The average attenuation factor of all tested items is 2.3%.

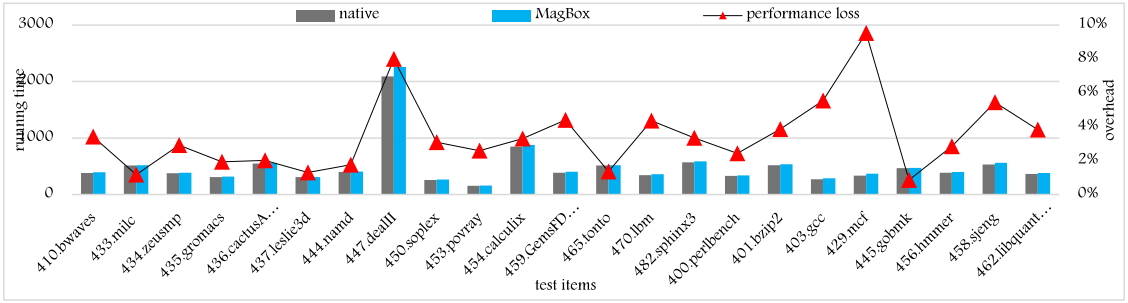


Fig. 11. SpecCPU2006 test results. The abscissa indicates the test items. The ordinate on the left indicates the test standard value, which corresponds to the bar graph; the ordinate on the right indicates the performance degradation factor, which corresponds to the line graph. The average attenuation factor of all tested items is 3.4%.

IOMeter. We use IOMeter to measure the performance of MagBox on I/O, which is shown in Fig. 12. The results show that the average overhead to *I/O Throughput* and *I/O Running Time* introduced by MagBox is 7.3% and 4.3%, respectively.

Web benchmark. We use Apache *httpd* to test MagBox's performance overhead to the network, as shown in Table 3. It shows the average processing time (ms) of *httpd* with different configurations. The results indicate that the overhead introduced by MagBox on the network are about 3% to 4%. Moreover, to observe the impact of MagBox on network throughput, we test the impact of MagBox on *Apache*, *Nginx* and *Lighthttpd* under different workloads. The experimental results are shown in Fig. 13. The results show that MagBox incurs 3%∼9% overhead in network throughput.

In summary, MagBox does not introduce excessive performance overhead to the OS, CPU, I/O, and networks. The main reason is that MagBox will not actively track all the jump branches. If and only when a risk function appears, MagBox migrates the risk function to a magic box to monitor and analyze its control flow transfers. This design can greatly reduce the redundancy of the control flow transfers to be tracked and detected.

In our experiments, we found the benchmarks mentioned above have no code probing activities. Therefore, there is no risk function to be tracked. To test the overhead introduced by tracking the risk function, we use binary rewriting technology to mark the benign functions (including library functions) in SpecCPU2006 as risk functions. After that, we test the output of SpecCPU2006 again. The test results are shown in Fig. 14. The results show that the performance overhead introduced by MagBox will increase as the number of the risk functions increases. When the number of the risk functions reaches 20, the performance overhead introduced by MagBox exceeds 15%.

MagBox will detect the code probing activities and migrate the risk function in which the probed code resides to a magic box. Next, the ICT instructions jumping out of the risk function will cause a system trap, whose legitimacy will be judged. The frequency of system traps directly affects the execution speed of

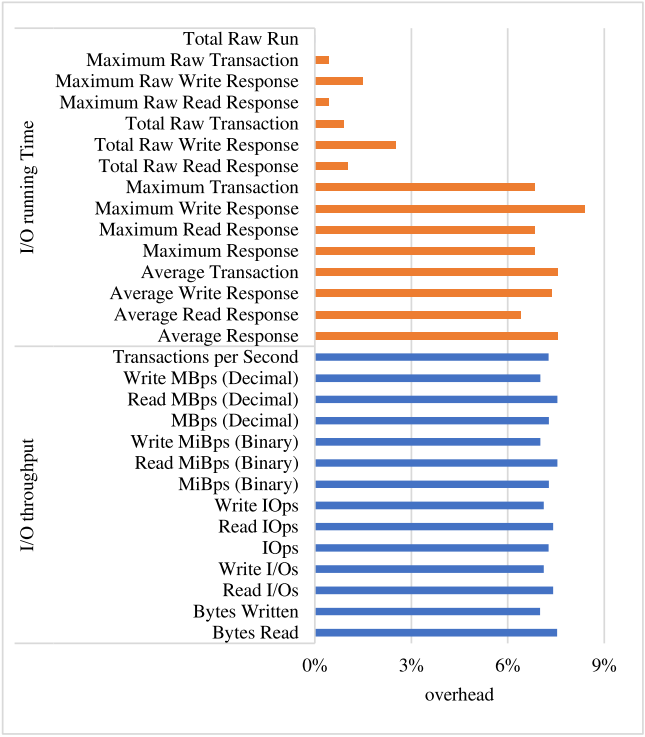
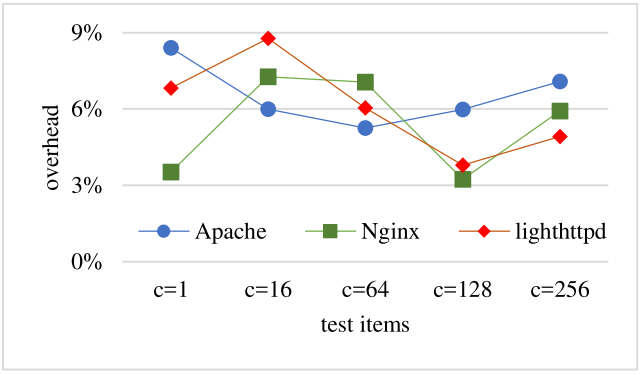


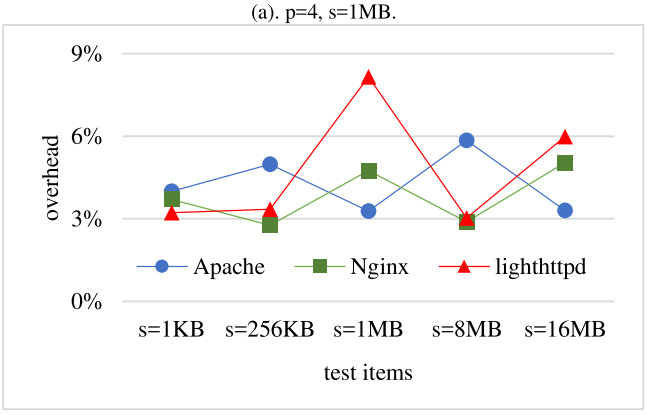
Fig. 12. IOMeter test results. The ordinate indicates the test items, and the abscissa indicates the overhead.

Table 3
Overhead incurred to *httpd* under MagBox with various numbers of worker processes and workloads. s = 1 MB.

Level	c = 1			c = 16			c = 64			c = 128			c = 256		
worker	Orig.	MagBox	Loss	Orig.	MagBox	Loss	Orig.	MagBox	Loss	Orig.	MagBox	Loss	Orig.	MagBox	Loss
p = 1	17.6	18.1	2.84%	15.6	15.9	1.92%	14.8	15.5	4.73%	14.8	15.3	3.38%	14.1	14.4	2.13%
p = 2	16.9	17.4	2.96%	11.3	11.6	2.65%	10.7	10.9	1.87%	10.8	11.2	3.70%	11.2	11.5	2.68%
p = 3	17.2	17.5	1.74%	10.1	10.9	7.92%	10.5	11.1	5.71%	11.2	12.1	8.04%	12.4	12.5	0.81%
p = 4	17.6	17.7	0.57%	9.9	10.4	5.05%	9.8	10.3	5.10%	10.6	10.7	0.94%	13.5	13.9	2.96%
p = 5	16.8	17.2	2.38%	10.2	10.6	3.92%	10.4	10.6	1.92%	10.4	10.7	2.88%	13.5	14.1	4.44%
p = 6	16.4	17.3	5.49%	10.3	10.5	1.94%	10.5	11.2	6.67%	9.9	10.2	3.03%	12.4	13	4.84%
p = 7	18.5	19.6	5.95%	9.8	10.4	6.12%	9.7	10.2	5.15%	10.7	11.2	4.67%	11.9	12.7	6.72%
p = 8	16.5	16.9	2.42%	10.3	10.4	0.97%	10.9	11	0.92%	10.3	10.9	5.83%	12.6	12.9	2.38%
Average			3.04%			3.81%			4.01%			4.06%			3.37%



(a). p=4, s=1MB.



(b). p=4, c=16.

Fig. 13. Network throughput overhead. a: Slowdown of number of requests per second incurred to web servers. b. Overhead of data transfer incurred to web servers.

the current process. The more risk functions, the more system traps triggered by tracking the control flow. Fortunately, the OS will not always be probed or attacked. In most scenarios, there is no risk functions in our OS. Taking a step back, if a risk function is detected, it is worth sacrificing part of the performance of a process to ensure the security of the process and the OS.

Micro benchmarks. To further observe the key factors affecting performance in MagBox, we introduce some microbenchmarks, as shown in Table 4. The meanings of the symbols in the table are as follows: *n_call*, *call address* in native space; *n_jump*, *jmp address* in native space; *n_ret*, *ret* in native space; *m_call*, *call address* that transfers control flow out of risk functions in magic box; *m_call**, *call *xx* that transfers control flow out of risk functions in magic box; *m_ret**, *ret* that transfers control flow to risk functions after executing *call ** in magic box; *m_jmp*, *jmp address* that transfers control flow in risk functions in magic box. It shows that a system trap takes much longer time than other normal code execution. In comparison, the EPT switch caused by *vmfunc* takes less time. In addition, when ICT instructions jump out of the risk function, they take significantly more time in the magic box than in the native space, which is caused by system traps. In contrast, the *call address* only needs to switch the EPT without triggering system traps. So, it executes faster. Compared with the *ret* in the native space, the *ret* in the magic box needs to cancel the breakpoint set at the stack in which the return address resides through a new system call. As a result, it takes more time. In magic box, *jmp** takes more time than *jmp address*, which is caused by system traps and control flow tracking. Compared with the *page walk* in the native OS, after the EPT is enabled, the page walk increases the access steps to the EPT, which leads to increased time-consuming. Although we introduce EPT, the time required by handling page faults does not increase significantly.

We count the number of control flow transfer instructions in typical applications, as shown in Table 5. The results show that the proportion of ICT instructions in all control flow transfer instructions is small. Another word, most control flow transfer instructions are legal, which are not available to attackers. Therefore, avoiding the legal instructions triggering system traps can greatly reduce the performance overhead. By switching EPT and adding a new system call, MagBox avoids the legal *call address*, *jmp address* and *ret* in the magic box triggering system traps.

Based on the above experiments and analysis, we can conclude that the system traps caused by MagBox is the main factor affecting the OS performance. They can be divided into 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 system traps unconditionally. According to our observations, the execution frequency of these instructions in different applications varies greatly. The system traps they cause will directly affect the running speed of the processes.

Conditional traps are triggered by the specific events set by MagBox, including breakpoint access, general protection exception, process creation, control flow jumps out of risk functions, etc. After MagBox handles the trap events, the OS will switch back to *guest* again. During the mode switching, the current process will be suspended, which increases the runtime overhead. The system trap and event handling caused by these events will slow down the execution speed of the process.

Performance on heavily loaded OS. Since the MagBox exclusive CPU when a system trap is triggered, it is necessary to know how MagBox performs when the OS is heavily loaded. Same with Buddy [16], we also use the *stress-ng* to control the CPU usage so that it ranges 5% to 99%. Then, we run SpecCPU2006 with and without MagBox, and record the runtime overhead introduced by MagBox, which is shown as Fig. 15. The results indicate MagBox is a practical solution in protecting CPU intensive programs.

Memory occupied. MagBox's code occupy less 1 MB memory. During its operation, it will allocate physical memory for magic box. The size of the physical memory depends on the size and number of risk functions. Fortunately, the memory occupied by

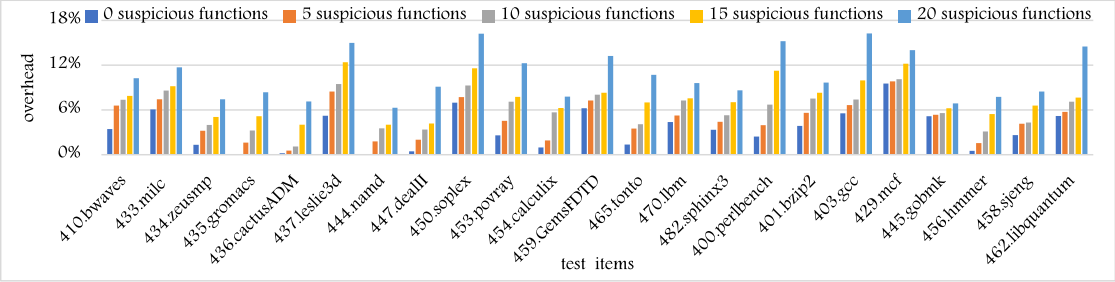


Fig. 14. Function migration test. The abscissa indicates the test items, and the ordinate indicates the performance degradation factor. Different colors represent different numbers of functions.

Table 4
Micro benchmarks (nanoseconds).

Native OS				OS with MagBox								
n_call	n_jmp	n_ret	page walk	vmfunc	system trap	m_call	m_ret	m_call*	m_ret*	m_jmp	m_jmp*	page walk
3.01	2.24	1.99	13.73	111.58	567.83	143.26	25.61	1229.73	27.39	2.27	1181.85	54.39

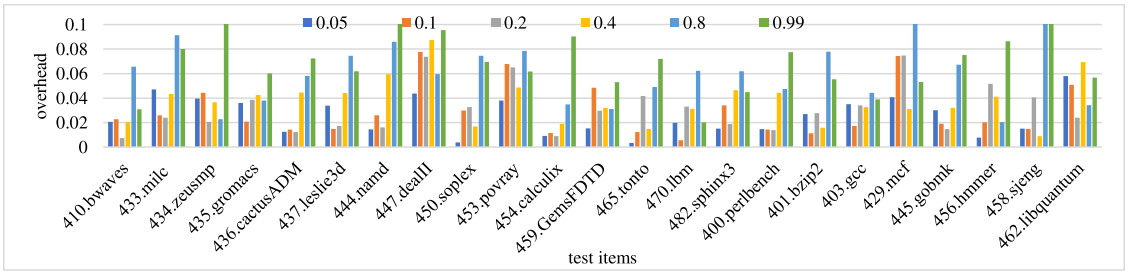


Fig. 15. Overhead of SPEC when CPU is in various load levels. The average overheads of CPU usage 5%∼99% are 2.5%, 2.9%, 3.1%, 3.8%, 6%, 6.8%.

Table 5
The number of control flow transfer instructions.

App	call addr	jmp addr	ret	call*	jmp*
httpd-2.4.37	34 843	15 194	5809	1224(2.1%)	142(0.2%)
redis-6.0.6	34 564	11 983	4875	558(1.1%)	621(1.2%)
nginx-1.6.2	5904	3395	1342	327(3%)	35(0.3%)

the risk functions in the entire memory space is not very large. For example, *skip_short_body*, the only known risk function in *wget*, only occupies 1.5 kB memory, which accounts for only 0.3% of the total *wget* code size. In addition, MagBox will allocate page tables for fake space. In MagBox, most of the fake space has only one page at each level of page tables. The entries in each level of page table point to their next level of page table, and all the entries in the current level of page table are the same. Therefore, we do not need to allocate too many page tables for fake space. In contrast, EPT occupies more memory. In our deployment (32 GB memory), a set of EPT occupies about 64 MB memory. If there is no risk function to be tracked, we only need one set of EPT; if there are risk functions to be tracked in the OS, we need two sets of EPT.

6.3. Comparison with existing methods

We compare MagBox with the existing CFI solutions according to the analysis method proposed in [46]. The results are shown in Fig. 16. There are 4 indicators, RP, CF, AP.A and AP.B. Except for RP, all other indicators are the qualitative result that analyzes the security method's defense principles, which is shown as metrics (1)∼(3). P^k_{app} and P^k_{lib} respectively represent the probability that the kth control flow transfer instruction can be tracked in the application and library. I^k_{app} and I^k_{lib} respectively represent the percentage of the kth control flow transfer instructions in applications and libraries.P^k_{attack} represents the probability that the kth attack scenario can be identified. $F\_I^k_{attack}$ and $B\_I^k_{attack}$ respectively represent the number of the illegal forward illegal instructions and the number of the illegal backward instructions in the kth attack scenario. $F\_I^k_{all}$ and $B\_I^k_{all}$ respectively represent the number of all tracked forward instructions and the number of all tracked backward instructions in the kth attack scenario.

CF refers to the control flow transfer instructions tracked by methods, which may be adopted by CRAs. The tracked instructions include *call *%register*, *call *(%register)*, *call *value(%register)*, *call *(%register, %register, value)*, *call *pointer*, *jmp *%register*, *jmp *(%register)*, *jmp *address(, %register, value)*, *ret*, *retn value*, and *retf value*. The more such instructions a method can track, the higher the CF score. Additionally, the instructions being tracked may be in shared libraries that have no source code and have been loaded into memory. A method that fails to track instructions in a library lowers its CF score. RP refers to the performance overhead reported in the paper. The lower the performance overhead, the higher the RP score. AP.F and AP.B are used to indicate the method's analysis precision for the control flow. Not all the control flow transfer instructions can be adopted by CRAs. In fact, they can only be used by CRAs in specific attack scenarios (such as memory leak scenarios). Therefore, the better a method can filter out potential attack scenarios, the higher its analysis precision is. In addition, the more effective the security strategies, the higher the AP.F and AP.B scores.

$$\mathrm{CF} = \sum_{k=1}^{N} (P^k_{app} \times I^k_{app} + P^k_{lib} \times I^k_{lib}) \tag{1}$$

$$\mathrm{AP.F} = \sum_{k=1}^{N} P^k_{attack} \times \frac{F\_I^k_{attack}}{F\_I^k_{all}} \tag{2}$$

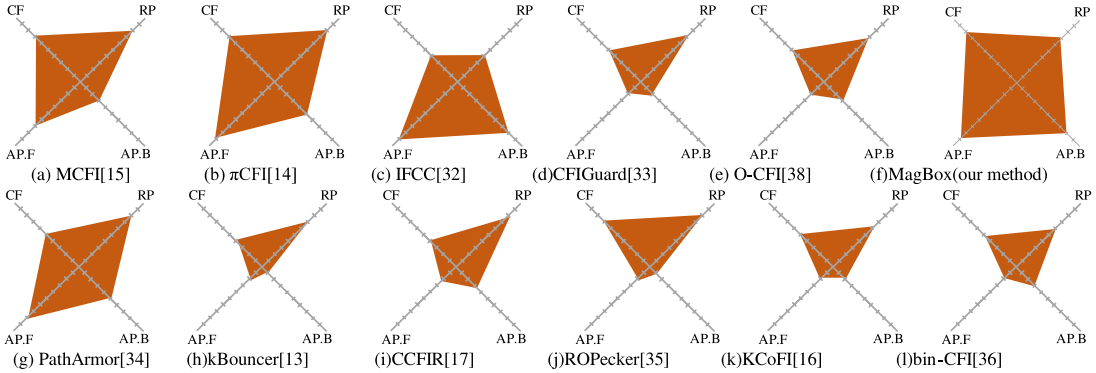


Fig. 16. Method comparison. CF: Control flow to be monitored. RP: The reported performance in papers. AP.F: The forward branch (such as *call* and *jmp*, etc.) analysis precision. AP.B: The back forward branch (return instructions *ret*) analysis precision [47–49].

$$\text{AP.B} = \sum_{k=1}^{N} P^{k}_{attack} \times \frac{B\_I^{k}_{attack}}{B\_I^{k}_{all}} \tag{3}$$

MagBox has some advantages in execution efficiency and defense effect. Because only the risk functions probed by the attacker are our targets, which can greatly reduce the number of instructions to be tracked. As a result, its performance overhead is not too high, and the AP.F and AP.B are perform well. Moreover, MagBox tracks control flow transfers at binary level dynamically, which reduces the complexity of legitimacy judgment. No matter what the control flow transfer instruction is, it will be detected when it is transferred outside or into the risk function, which improves its CF score. In a word, whether it is ROP, JOP, LOP, or COOP, as long as it is under fine-grained ASLR, it needs to probe the code, which can be detected and analyzed by MagBox.

The compiler-based methods, such as IFCC, πCFI, and MCFI, require the support of source code, which leads to the protection failure on the loaded libraries. This characteristics can negatively impact on their CF, AP.F and AP.B. The methods using hardware-assisted techniques can achieve better results with less overhead, such as O-CFI using MPX and PathArmor using LBR. ROPecker detects an ROP attack at run-time by checking the presence of a sufficiently long chain of gadgets in past and future execution flow, with the assistance of the taken branches recorded in the LBR and an efficient technique combining offline analysis with run-time emulation [50]. However, it is invalid to JOP, LOP, and COOP, etc. CFIGuard detects all indirect jumps, and it relies on a high-precision CFG. For shared libraries that do not contain source code, the jump relationship between code blocks is not clear, which may be changed with the input and conditions. A low-precision CFG will reduce the AP.B and AP.F of CFIGuard.

7. Conclusions

To mitigate CRAs, this paper proposes MagBox. Under the protection of ASLR, attackers need to use code probing technology to obtain code information, such as code forms and code addresses. Detecting and preventing code probing can mitigate CRAs. MagBox uses the fake space mechanism to detect the attacker's code probing. Using the attacker's probing ability, MagBox can identify and locate risk functions. Then, the risk functions will be migrated to a magic box with specific space structure and memory permissions. After that, all the instructions that jump out of the risk function will be monitored and analyzed in real time. Experiments show that MagBox has good defense effects on CRAs, and it introduces low overhead to the OS and CPU.

However, MagBox still has some limitations. First, MagBox is only valid in user space. Compared with user space, the kernel code segment is completely shared. Moreover, there are many data structures containing function pointers in the kernel space. The attacker can obtain the address of the kernel code by reading kernel data, and he does not need to probe the kernel code frequently. This can reduce the risk of the probed code being exposed, which avoids the illegal control flow being tracked and analyzed by MagBox. Second, MagBox only supports Linux and the processors with X86 architecture. In future research, we will try to promote MagBox to Windows and deploy it on Arm processors.

CRediT authorship contribution statement

YongGang Li: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing, Project administration. **GuoYuan Lin:** Writing – review & editing, Supervision. **Yeh-Ching Chung:** Writing – review & editing, Supervision. **YaoWen Ma:** Formal analysis, Investigation. **Yi Lu:** Formal analysis, Investigation. **Yu Bao:** Data curation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

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

References

[1] Y.G. Li, Y.C. Chung, Y. Bao, et al., KPointer: Keep the code pointers on the stack point to the right code, Comput. Secur. (2022) 102781.
[2] Stephen Checkoway, et al., Return-oriented programming without returns, in: Proc. the 17th ACM Conference on Computer and Communications Security, 2010, pp. 559–572.
[3] B. Lan, Y. Li, et al., Loop-oriented programming: a new code reuse attack to bypass modern defenses, in: Proc. IEEE Trustcom/BigDataSE/ISPA, 2015, pp. 190–197.
[4] R. Strackx, Y. Younan, et al., Breaking the memory secrecy assumption, in: Proc. the Second European Workshop on System Security, 2009, pp. 1–8.
[5] W.L. Mow, S.K. Huang, H.C. Hsiao, LAEG: Leak-based AEG using dynamic binary analysis to defeat ASLR, in: Proc. IEEE Conference on Dependable and Secure Computing, DSC, 2022, pp. 1–8.
[6] K. Lu, C. Song, B. Lee, S.P. Chung, et al., ASLR-guard: Stopping address space leakage for code reuse attacks, in: Proc. the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 280–291.
[7] A. Bittau, A. Belay, et al., Hacking blind, in: Proc. the IEEE Symposium on Security and Privacy, IEEE, 2016, pp. 227–242.
[8] Kangjie Lu, et al., How to make ASLR win the clone wars: Runtime re-randomization, in: Proc. NDSS, 2016.

[9] B. Gras, K. Razavi, E. Bosman, et al., ASLR on the line: Practical cache attacks on the MMU, NDSS 17 (2017) 26.
[10] J. Li, L. Chen, G. Shi, et al., ABCFI: Fast and lightweight fine-grained hardware-assisted control-flow integrity, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (11) (2020) 3165–3176.
[11] P. Muntean, M. Neumayer, Z. Lin, et al., Analyzing control flow integrity with LLVM-CFI, in: Proc. the 35th Annual Computer Security Applications Conference, 2019, pp. 584–597.
[12] R. Ding, C. Qian, C. Song, et al., Efficient protection of path-sensitive control security, in: Pro. the 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 131–148.
[13] X. Ge, W. Cui, et al., Griffin: Guarding control flows using intel processor trace, ACM SIGPLAN Not. 52 (4) (2017) 585–598.
[14] B. Niu, G. Tan, Modular control-flow integrity, in: Proc. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 577–587.
[15] C. Tice, T. Roeder, P. Collingbourne, Enforcing forward-edge control-flow integrity in {gcc} & {llvm}, in: Proc. 23rd {USENIX} Security Symposium ({USENIX} Security 14), 2014, pp. 941–955.
[16] K. Lu, M. Xu, C. Song, et al., Stopping memory disclosures via diversification and replicated execution, IEEE Trans. Dependable Secure Comput. 18 (1) (2018) 160–173.
[17] Z. Huang, T. Zheng, Y. Shi, A. Li, A dynamic detection method against ROP and JOP, in: Proc. 2012 International Conference on Systems and Informatics, 2012, pp. 1072–1077.
[18] Felix Schuster, et al., Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications, in: Proc. IEEE Symposium on Security and Privacy, 2015.
[19] P. Chen, H. Xiao, X. Shen, DROP: Detecting return-oriented programming malicious code, in: Proc. International Conference on Information Systems Security, 2009, pp. 163–177.
[20] M. Kayaalp, T. Schmitt, et al., SCRAP: Architecture for signature-based protection from code reuse attacks, in: Proc. IEEE 19th International Symposium on High Performance Computer Architecture, HPCA, 2013, pp. 258–269.
[21] V. Pappas, M. Polychronakis, et al., Transparent {ROP} exploit mitigation using indirect branch tracing, in: Proc. 22nd {USENIX} Security Symposium, 2013, pp. 447–462.
[22] B. Niu, G. Tan, Per-input control-flow integrity, in: Proc. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 914–926.
[23] P. Yuan, Q. Zeng, X. Ding, Hardware-assisted fine-grained code-reuse attack detection, in: Proc. International Symposium on Recent Advances in Intrusion Detection, 2015, pp. 66–85.
[24] Hong Hu, et al., Enforcing unique code target property for control-flow integrity, in: Pro. the ACM SIGSAC Conference on Computer and Communications Security, 2018.
[25] J. Criswell, N. Dautenhahn, KCoFI: Complete control-flow integrity for commodity operating system kernels, in: Proc. IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 292–307.
[26] C. Zhang, T. Wei, Z. Chen, L. Duan, et al., Practical control flow integrity and randomization for binary executables, in: Proc. 2013 IEEE Symposium on Security and Privacy, 2013, pp. 559–573.
[27] M. Zhang, R. Sekar, Control flow integrity for {cots} binaries, in: Proc. 22nd {USENIX} Security Symposium, 2013, pp. 337–352.
[28] V. Mohan, P. Larsen, S. Brunthaler, et al., Opaque control-flow integrity, in: Proc. NDSS, Vol. 26, 2015, pp. 27–30.
[29] S. Yoo, J. Park, S. Kim, et al., {In-kernel}{control-flow} integrity on commodity {oses} using {arm} pointer authentication, in: Proc. USENIX Security Symposium (USENIX Security 22), 2022, pp. 89–106.
[30] A. Gupta, S. Kerr, et al., Marlin: A fine grained randomization approach to defend against ROP attacks, in: Proc. International Conference on Network and System Security, 2013, pp. 293–306.
[31] J. Hiser, A. Nguyen-Tuong, et al., ILR: Where'd my gadgets go? in: Proc. IEEE Symposium on Security and Privacy, 2012, pp. 571–585.
[32] K.Z. Snow, F. Monrose, et al., Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, in: Proc. 2013 IEEE Symposium on Security and Privacy, 2013, pp. 574–588.
[33] Davi. Lucas, et al., Isomeron: Code randomization resilient to (just-in-time) return-oriented programming, in: Proc. NDSS, 2015.
[34] J. Seibert, H. Okhravi, et al., Information leaks without memory disclosures: Remote side channel attacks on diversified code, in: Proc. Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 54–65.
[35] C. Curtsinger, E.D. Berger, Stabilizer: Statistically sound performance evaluation, Proc. ACM SIGARCH Comput. Archit. News 41 (1) (2013) 219–228.
[36] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, G. Portokalidis, Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard, in: Proc. 23rd {USENIX} Security Symposium, 2014, pp. 417–432.
[37] J. Ying, R. Hou, L. Zhao, et al., CPP: A lightweight memory page management extension to prevent code pointer leakage, J. Syst. Archit. (130) (2022) 102679.
[38] A. Oikonomopoulos, E. Athanasopoulos, et al., Poking holes in information hiding, in: Proc. the 25th {USENIX} Security Symposium, 2016, pp. 121–138.
[39] R. Gawlik, B. Kollenda, P. Koppe, et al., Enabling client-side crash-resistance to overcome diversification and information hiding, in: Proc. the NDSS, Vol. 16, 2016, pp. 21–24.
[40] https://www.intel.cn/content/www/cn/zh/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.html.
[41] L. Zhang, D. Choffnes, D. Levin, et al., Analysis of SSL certificate reissues and revocations in the wake of heartbleed, in: Pro. the Conference on Internet Measurement Conference, 2014, pp. 489–502.
[42] H. Hu, S. Shinde, S. Adrian, et al., Data-oriented programming: On the expressiveness of non-control data attacks, in: Proc. IEEE Symposium on Security and Privacy, SP, 2016, pp. 969–986.
[43] Y.G. Li, Y.C. Chung, K. Hwang, et al., Virtual wall: Filtering rootkit attacks to protect linux kernel functions, IEEE Trans. Comput. 70 (10) (2021) 1640–1653.
[44] Y. Guo, L. Chen, G. Shi, Function-oriented programming: A new class of code reuse attack in c applications, in: Pro. IEEE Conference on Communications and Network Security, CNS, 2018, pp. 1–9.
[45] J. Salwan, ROPgadget–Gadgets Finder and Auto-Roper. http://shell-storm.org/project/ROPgadget.
[46] N. Burow, S.A. Carr, J. Nash, et al., Control-flow integrity: Precision, security, and performance, ACM Comput. Surv. 50 (1) (2017) 1–33.
[47] V. Van der Veen, D. Andriesse, E. Göktaş, et al., Practical context-sensitive CFI, in: Proc. the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 927–940.
[48] M. Zhang, R. Sekar, Control flow integrity for {cots} binaries, in: Proc. 22nd {USENIX} Security Symposium, 2013, pp. 337–352.
[49] V. Mohan, et al., Opaque control-flow integrity, in: NDSS Symposium, 2015.
[50] Y. Cheng, Z. Zhou, M. Yu, X. Ding, et al., ROPecker: A generic and practical approach for defending against ROP attack, in: Proc. 21th NDSS, 2014.

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. His research interests include computer architecture, virtualization principle, cloud computing, and system security.

Guo-Yuan Lin, received the PhD degree from the Nanjing University in 2011. Now he is the deputy dean of with the School of Computer Science and Technology in the China University of Mining and Technology. He has long been engaged in the research of cyberspace security, information security, and system security.

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

Yao-Wen Ma, is a graduate student at the School of Computer Science and Technology in the China University of Mining and Technology. His research interests include container architecture and cloud computing.

Yi Lu, is a graduate student at the School of Computer Science and Technology in the China University of Mining and Technology. His research interests include code optimization and cloud computing.

Yu Bao, received the PhD degree from Tongji University in 2011. Now, he is a staff engineer at security Department of Computer Science and Information Technology Institute, China University of Mining and teach. His research includes information security and privacy in AI distributed network and cyber security in IoT.