Randomize the Running Function When It Is Disclosed

YongGang Li, Yu Bao, and Yeh-Ching Chung, *Senior Member, IEEE*

***Abstract*—Address space layout randomization (ASLR) can hide code addresses, which has been widely adopted by security solutions. However, code probes can bypass it. In real attack scenarios, a single code probe can only obtain very limited code information instead of the information of the entire code segment. So, randomizing the entire code segment is unnecessary. How to minimize the size of the randomized object is a key to reducing the complexity and overhead for ASLR methods. Moreover, ASLR needs to be completed between the time after code probe occurs and before the probed code is used by attackers, otherwise it is meaningless. How to select an appropriate randomization time point is a basic condition for achieving effective address hiding. In this paper, we propose a runtime partial randomization method RandFun. It only randomizes the probed function with parallel threads. And the randomization is performed when and only when potential code probes are detected. In addition, RandFun can protect the probed code from being used as gadgets, whether during or after randomization. Experiments and analysis show RandFun has a good defense effect on code probes and only introduces 1.6% overhead to CPU.**

***Index Terms*—Code probes, code reuse attacks, control flow, access control, operating system.**

I. Introduction

THE deployment of CRAs (code reuse attacks) needs to obtain specific code snippets and their addresses. ASLR can hide code addresses, so that the specific code snippets cannot be located. However, code probes can bypass ASLR [2]. The existing code probes include allocation oracle [1], process clone [7], arbitrary read [3], data leakage [4], arbitrary jump [5], and side-channel leakage [6], etc.

To defend against code probes, re-randomization [8] is proposed. In fact, it only makes sense to randomize the probed code during the period after code probes occur and before CRAs are deployed. Randomizing the non-probed code is meaningless. The ideal re-randomization should answer how to choose the right time to randomize the disclosed target, which directly affects its effectiveness and efficiency.

Manuscript received 3 July 2023; revised 28 January 2024; accepted 21 February 2024. Date of publication 4 March 2024; date of current version 10 May 2024. This work was supported by the Fundamental Research Funds for the Central Universities under Grant 2023QN1078. Recommended for acceptance by M. Correia. *(Corresponding author: YongGang Li.)*

YongGang Li and Yu Bao are with the China University of Mining and Technology, Wudaokou, Haidian 100083 China (e-mail: lygzr@mail.ustc.edu.cn).

Yeh-Ching Chung is with the Chinese University of Hong Kong in Shenzhen, Shenzhen, 518172 China.

Digital Object Identifier 10.1109/TC.2024.3371776

The first challenge faced by re-randomization is how to choose the disclosed target to randomize. In the presence of fine-grained ASLR, a single probe or leakage can only expose a small amount of code information. Therefore, it is unnecessary to re-randomize the entire code segment. The granularity of randomization and the size of the target determine ASLR's complexity. Performing fine-grained re-randomization for the entire code segment can increase the suspend time of the process. For example, Shuffler introduces over 50% overhead to xalancbmk [9]. The reason is such a method requires a lot of extra effort to ensure the control flow can jump to the right location after randomization. To profile the complex call graph of the randomized objects, existing methods have to rely on source code, which is the basic reason why most methods only choose the objects with source code as their protection targets.

The second challenge faced by re-randomization is how to select the time point of randomization. Traditional ASLR randomizes targets via the modified compiler, loader or linker. While, the code layout remains unchanged until the process ends. An attacker can bypass such methods by probing the binary code. In contrast, re-randomization randomizes the code at a specific point during the process running. For the periodic ASLR methods, a large randomization interval can leave sufficient attack windows for attackers, while a small interval can incur significant overhead. For the runtime methods, they randomize the target when the code with specific characteristics is executed, such as the system call *write*. However, overly loose randomization conditions can also introduce significant overhead.

Faced with the two challenges, this paper proposes a novel method RandFun, whose protection target is the running binary code. It needs to address four issues. (1) How to select the object that should be randomized? (2) When the selected object needs to be randomized? (3) How to reduce the process suspending time caused by randomization? (4) How to ensure the security of control flows during and after randomization.

To address the issues (1) and (2), RandFun builds a mechanism to perceive code probes in real time. Next, it migrates the probed function to a new space to ensure it can be called legally. Meanwhile, a parallel thread is started to randomize the probed function, which can address the issue (3). During and after the randomization, the control flow jumping to the probed function will be tracked and detected, which can address the issue (4).

The motivation of this paper is to propose a runtime randomization method for the running process without source code.

Unlike existing methods, we only randomize the functions (rather than entire code segments) that may be disclosed in the potential attack scenarios (rather than all execution scenarios). Such designs can reduce unnecessary randomization and avoid profiling complex control flow graphs for the entire code segment. In summary, the contributions of this paper are as follows:

1) Propose a mechanism to perceive code probes, which are the events triggering randomization.
2) Propose a real runtime randomization method. It only randomizes the code blocks within the probed function, not the entire code segment. The randomization activity and the protected process are parallel, which can reduce the running delay of the protected process.
3) Propose a control flow protection mechanism. It can prevent the probed functions and code blocks from being used as gadgets during and after randomization.
4) Implement RandFun in Linux. To the best of our knowledge, RandFun is the first runtime ASLR for the probed targets and it only introduce 1.6% overhead to CPU.

II. Background and Related Works

CRAs are a series of control flow hijacking methods. They do not require injecting code into the attacked object, but instead leverage existing code to deploy attacks. CRAs use memory vulnerabilities (such as stack overflow vulnerabilities) to tamper with control flows, and redirect them to the selected malicious payloads (gadgets) instead of the original code. Gadgets are the code snippets located in the code segment of a process or library. They contain control flow transfer instructions including *call *pointer/*register*, *jmp *pointer/*register*, and *ret*, which can transfer control flows to the next gadget. Currently, the most commonly used method for defending CRAs is the control flow integrity (CFI) method. It sets checkpoints at the locations where the control flow transfers occur. However, for the closed-source software, existing methods cannot set effective checkpoints through source code analysis or compilation, resulting in an inability to deploy.

For the closed-source software, illegal control flows can be detected by monitoring and analyzing their behaviors. Hardware-assisted virtualization technologies EPT (Extended Page Tables) and VMX (Virtual Machine Extension) can be used to efficiently track and detect the behavior of running processes. EPT is a memory virtualization technology that can control page permissions and mapped areas. VMX is a control technology that can capture and manage specific events in the OS (such as process switching and interrupts). Both have been widely used in the field of control flow protection and have achieved good results.

Deploying CRAs requires obtaining the code snippets that conforms to the gadget format in advance. ASLR can change the code layout, thereby hiding code addresses from attackers. To defeat ASLR, code probes are proposed. The essence of code probes is that attackers obtain code addresses and code forms through memory access. In general, code-reading-based probes can directly obtain the code forms. Moreover, existing code probes can also obtain mapped code areas, specific code addresses, and even indirectly obtain code snippets that conform to the gadget format. Taking the vulnerability CVE-2015-7547 as an example, it can be used to deploy arbitrary-jump probes. In this probe, attackers can use the vulnerability to tamper with the original return address, causing it to jump to an arbitrary location. Afterwards, attackers continue to fill the stack with multiple return addresses, which point to a notification module (such as a net function). Therefore, if the hijacked control flow jumps to a code snippet containing the instruction *ret*, attackers can know that the executed code containing *ret*.

In fact, under the protection of randomization (especially fine-grained randomization), code probes have become the first step in deploying CRAs. Defending code probes can effectively mitigate the threat posed by CRAs. The methods defending against code probes include memory isolation and runtime ASLR. In this section, we introduce them separately.

A. Memory Isolation

Memory isolation divides the memory into normal area and safe area. The protected objects are placed in the safe area, and only the specific code can access them. TDI [10] isolates memory objects of different colors in separate memory arenas. MemSentry [11] is a domain-based method that uses Intel MPK to protect the sensitive data, and it only allows the data access when execution has switched to the right domain. Datashield [12] separates the memory into a precisely protected region for sensitive data and a coarsely protected region for non-sensitive data. ConfLLVM [13] is a compiler-based method, which allows classifying data into public and private. MemCat [14] tries to identify the data that can be controlled by adversaries using compile-time policy, and it allocates those objects on a separate heap/stack. SeCage [15] protects the target data via EPT switching. Type-After-Type splits available memory into separate pools for each type of heap data [16].

In fact, the most serious challenge is not how to isolate the target objects, but how to determine which objects need to be isolated. It is an important premise for current methods that the defender knows which data or code are vulnerable in advance. However, the execution logic of the protected object, especially for the closed-source projects, is not always known. Therefore, the existing isolation methods still have obvious weaknesses in terms of protection effect and deployment scope.

B. ASLR

The existing ASLR can be divided into data ASLR and code ASLR. Data ASLR randomizes the representation of data in memory to prevent deterministic corruption of data or plaintext information leakage [10]. CoDaRR [17] continuously re-randomizes the masks used in loads and stores, and re-masks all memory objects to remain transparent *w.r.t* program execution. PT-Rand [18] randomizes the location of page tables to ensure the location of page tables is not leaked. The main idea behind ASLR-Guard [24] is to provide a secure storage for code pointers and encode the code pointers when they are treated as data.

Compared with data ASLR, more code ASLR methods have been proposed. The randomization granularities include code pages, functions, basic blocks, and instructions. SafeHidden

[19] continuously re-randomizes the locations of safe areas. Shuffler [9] defend against blind ROP and JIT-ROP in user space. CodeArmor [20] is another user-space re-randomization method, and its efficiency is better because of page remapping. Remix [21] randomly shuffles basic blocks within their respective functions. TASR [22] re-randomizes the memory layout during runtime before the adversary can take advantage of any stolen knowledge. Adelie enables efficient continuous KASLR for modules by using the PIC [23]. RUNTIMEASLR [7] prevents clone-probe attacks by re-randomizing the address space of every child after *fork()*. CCR [25] relies on compiler–rewriter cooperation to enable fast and robust fine-grained code randomization on end-user systems. Mixr [26] works on software/libraries without access to their source code. Reranz [31] uses parallel processes to randomize the entire code segment, which can reduce the running delay.

Almost all ASLR methods randomize the entire segment, even most of the object are unknown to the adversary. This means a large number of addressing operations need to be modified to ensure the code can be correctly called after randomization. This is a huge challenge for existing methods, especially for the runtime randomization, which incurs significant overhead. In addition, complex randomization steps can suspend the process for a long time, thus causing a delay that cannot be ignored.

III. Assumptions and Threat Models

First, we assume the address space of the target process have been randomized with the function granularity during compiling or loading, and adversaries must probe code to find gadgets, which is similar with Buddy [27]. Therefore, adversaries can only obtain the memory layout of a single function at most via a single leaked pointer (such as the return address). Second, we assume adversaries can probe the applications in user space with the existing probe technologies [1], [2], [3], [4], [5], [6]. Third, we assume adversaries can exploit the memory corruption (such as the overflow vulnerabilities) to hijack control flows. The threat model in this paper focuses on the following 5 memory probes:

Vector 1: Allocation Oracle [1]. It uses memory allocation functions, such as *malloc*, to allocate an area, and the returned result will reveal whether the area has been mapped.

Vector 2: Clone Probe [2]. It uses the cloned process to probe the memory layout of the parent process, which can avoid the parent process's crash caused by illegal access.

Vector 3: Arbitrary Read [3]. It can arbitrarily read code and data. Moreover, Data Leakage [4] can read the relative offset in PLT (Procedure Linkage Table) to get the randomized address of GOT (Global Offset Table), which stores all the library function addresses needed by the current process.

Vector 4: Arbitrary Jump [5]. It can redirect control flows to any position through arbitrary write. Then the available gadgets can be located by analyzing the crash information.

Vector 5: Side-channel Leakage [6]. It uses the cache hit and cache miss in the page tables at all levels to crack the 12^{th} to 47^{th} bits of the virtual address step by step.

IV. System Design

Existing ASLR methods randomize the entire code segment during code compilation, process loading, or process running. To ensure the control flow can still jump to the right location after code randomization, a precise control flow graph is necessary. However, the larger the size of the randomized code, the more complex the control flow graph. Therefore, existing methods can only construct precise control flow graphs by analyzing the source code. In addition, randomization operations must be completed after the occurrence of code probes and before the probed code is executed, otherwise the randomization is invalid or meaningless. RandFun aims to randomize the probed functions when code probes occur and protect the control flow jumping into them. But, why don't we directly block the activities with probe risks? The reasons are as follows:

First, directly blocking the risk activity may lead to conflicts. In fact, the current risk activity is not necessarily a code probe. For example, code reading can be triggered either by a code probe or by a debugger. Second, directly blocking the risk activity cannot prevent the specific payloads from probing code or hijacking control flows in the next round of attacks, even if it is a real code probe. Some malicious payloads may have been known by an attacker before the current probe was detected. For example, for a known stack overflow vulnerability, an attacker only needs to design inputs to hijack the return control flow without any code probe. Third, some gadgets may have been known before the risk activity is detected. For example, the illegal jumps may have performed multiple times, and a target code snippet containing *ret* has been located before the current activity triggers an exception signal. In practise, the undetected code snippets that have been selected by attackers are more dangerous. Fortunately, the number of such payloads is not large enough to build a complete gadget chain. Attackers need code probes to obtain more payloads and chain them with the undetected one. From the perspective of attack principles, starting with the probed code and retroactively checking the executed code blocks can help to discover more payloads that can be maliciously exploited by attackers but unknown for us.

Considering the above issues, RandFun does not directly prevent the code probes, but rather randomizes the probed functions at runtime and checks the control flow jumping to them. To achieve these goals, RandFun needs to complete the following tasks.

1) RandFun builds a mechanism to detect code probes and filter out the function with exposure risk. In the presence of fine-grained ASLR, adversaries need to perform code probes to get code forms or addresses [27]. However, code probes are mixed with normal code accesses, which makes them difficult to be identified. Fortunately, there are subtle or obvious differences between legal code accesses and illegal code probes, which will be reflected in execution context and behavior characteristics. First, the running processes generally do not read their own code and library code, while some code probes require reading code to filter out enough gadgets. Second, legal code does not access unmapped areas, while some code probes inevitably access unmapped areas. Third, legal code execution rarely triggers

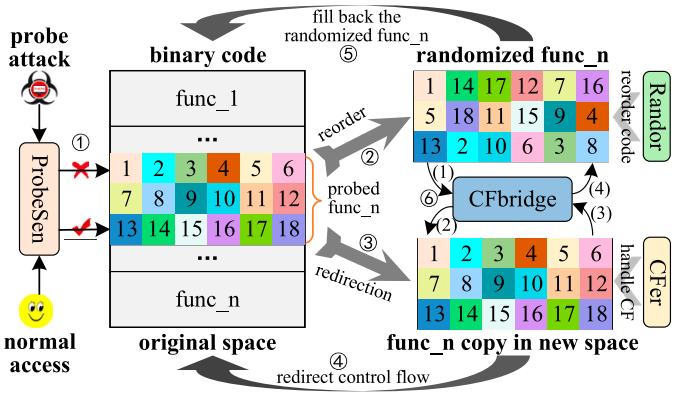

Fig. 1. The overall design of RandFun.

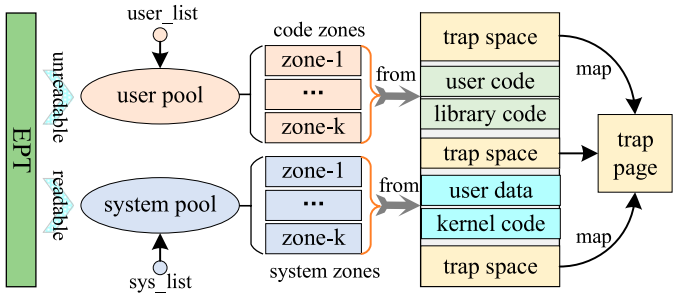

Fig. 2. The mechanism perceiving code probes.

the exception signal SIGILL, while some code probes (such as arbitrary jumps) have a high probability of triggering SIGILL. Fourth, some code probes (such as side channel-based probes) regularly access specific spaces, which is rare for the legal code. Although there is no strict boundary between code probes and legal code access, we can still use the differences between them to screen out the potential code probes.

2) RandFun randomizes the probed code during the process running. Directly manipulating the running code may cause a process crash. To avoid this problem, RandFun copies the probed code into kernel space for randomization. After that, the randomized code will be backfilled into the original space.

3) RandFun protects the probed code snippets from being used as gadgets. In the period during code randomization, the risk that the probed code is used as a gadget is much higher than that of the un-probed code. Because the process is not suspended during randomization. At this point, the probed code may have been known by the attacker, and it can be called as a gadget. To chain the gadget, the attacker must break the original code logic or data logic. This feature can be used to identify illegal control flows. Moreover, RandFun also prevents the probed function from being used as a function-granularity gadget, such as COOP [28].

RandFun consists of ProbeSen, Randor, CFer, and CFbridge, as shown in Fig. 1. First, ProbeSen perceives code probes and finds out probed functions (①). Next, Randor migrates the probed functions to a new space and copy them into kernel for randomization (②). Meanwhile, it fills *int3* to the original function location to capture the control flow transferred to the probed function. CFer handles the control flow transferred to the probed function during randomization. It has two tasks: one is to check the legitimacy of control flow transfers (③), and the other is to transfer the legal control flow to its destination (④). Finally, CFbridge backfills the randomized function into the original space (⑤) and check the legitimacy of the calls to the randomized function (⑥). It should be noted that there may exist return addresses on the stack that point to the migrated function in the new space after the randomized function has been backfilled into the original space. Scanning the entire stack to find them is both time-consuming and inaccurate. To address this issue, CFbridge continues to capture the control flow that jumps to the migrated function in the new space. It redirects the return control flow to the randomized function in the original space ((1)-(4)) after detection.

The above designs require the capabilities of tracking and controlling process behaviors. To achieve this purpose, we combine VMX root and VMX non-root to divide the running OS (operating system) into host and guest. In normal scenarios, the OS runs in the guest. When a specific event occurs, the running mode will switch to the host, which is called a system trap. Combined with EPT and VMX, various system trap events can be configured, which include process switching, breakpoints, execution of specific instructions (such as *int3*), interruption, single-step debug, and protection exception, etc. We can even rewrite the CPU context by modifying the fields in VMCS (virtual machine control structure).

V. System Implementation

A. Perceive Code Probes

Since the code segment is unwritable, adversaries probe code by reading it. XOM [38] disables the reading permissions of code pages, thus preventing code leakage. However, code pages are not completely loaded into memory at one time, which makes it impossible to set all code pages as unreadable. Existing methods have to track page allocation and adjust the permission page by page, which introduce obvious overhead. The component ProbeSen (shown in the Fig. 1) is designed to perceive code probes.

1) Manage Code Permissions: We rebuild the memory allocation system *buddy system* of Linux, as shown in Fig. 2. In UMA architecture, *zone_list* points to the *zones* that contain all pages. In the new *buddy system*, the original *zones* are divided into two categories, *code zones* and *system zones*. The two *zones* are abstracted into two pools, *user pool* and *system pool*, which are pointed by *user_list* and *sys_list* respectively. All user code pages are allocated from the *user pool*. Other code and data, such as user data and kernel code, are allocated from the *system pool*. All pages in *user pool* are unreadable, which can be achieved by setting EPT entries. In summary, the new *buddy system* can set all code pages in user space to be unreadable at one time, which avoids page tracking during process running.

Note that we cannot use EPT to set a page to be unreadable and writable at the same time. Otherwise, an EPT exception will be triggered. However, the code page must be writable when it

is being loaded, while the pages in user pool are unreadable. To solve this conflict, we enable an EPT_switch, in which all pages are readable, writable, and executable. If and only if a code page fault occurs in user space, the added VMX instruction *vmfunc* at the head of the function *filemap_fault* (a function that loads code from an ELF file into memory) switch the current EPT to EPT_switch. Then, the code page is readable and writable. When *filemap_ fault* returns, *vmfunc* is executed again to switch back to the original EPT. At this point, the code page has been loaded into memory, and the code page is unreadable and unwritable.

The new buddy system can preset all code pages to be unreadable, which can perceive the code reading (Vector 3). When the code probe based on code reading occurs, an EPT exception will be triggered. Next, the reading permission of the current page is enabled. Then, the *EFLAGS.TF* in VMCS will be set to 1, which can put the processor into the single-step debug mode. Then, every code reading can be captured. The function that has been read will be marked as a probed function. This design can perceive the Vector 3.

In addition to code reading, adversaries can also deduce the code forms through the exception caused by code execution (such as Vector 4). In the presence of fine-grained ASLR, the adversary's arbitrary jump probably jumps to unmapped areas or illegal instructions. The former will trigger the signal SIGSGEV, and the latter will trigger the signal SIGILL.

2) Capture Signals: In our design, the system calls *signal* and *sigaction* will be modified to capture the signals SIGSGEV and SIGILL. Theoretically, the adversary may have performed several illegal jumps before a signal is triggered, and its previous jumps did not trigger any exceptions. Therefore, we need to detect whether these control flow transfers are caused by code probes. After the SIGSGEV or SIGILL is captured, they use the security strategies (described in Section V-C) to analyze the code blocks recorded by LBR (Last Branch Record). This design can perceive Vector 4.

3) Set Trap Spaces: To perceive Vector 1 and Vector 5, the trap space mechanism is proposed. In Linux, users have up to 128TB address space. In practice, only a small part is in use. Most of the space is unmapped. We allocate additional address spaces for the target process when it is created. For example, the original code size is 1MB, and it is just one of the 10^6 mapped spaces in the trap mechanism. If the adversary does not have any prior knowledge of the code, the probability that the control flow is transferred to the real code area at one time is only $1/10^6$. The additional areas mapped by us are called trap spaces. They will be mapped to a same unreadable, non-writable, and non-executable page called trap page, which is achieved by modifying the EPT entries.

Allocation oracle [3] (Vector 1) probe the mapped spaces through the results of memory allocation, which can locate the hidden area. Under trap space mechanism, allocation oracle can still get a lot of mapped areas, but it cannot filter out the real code area from them, which leads to a probe failure.

In contrast, side channel leakage [4] (Vector 5) can accurately predict the $12^{\text{th}} \sim 47^{\text{th}}$ bits of the target address by the time difference between TLB hit and TLB miss. To observe the status of TLB, adversaries repeatedly evict or fill the content in TLB via flush+reload, EVICT+TIME or PRIME+PROBE [40]. To crack the $12^{\text{th}} \sim 14^{\text{th}}$, $21^{\text{st}} \sim 23^{\text{rd}}$, $30^{\text{th}} \sim 32^{\text{nd}}$, and $39^{\text{th}} \sim 41^{\text{st}}$ bits, the memory separated from the virtual address by n*4KB, n*2MB, n*1GB, and n*512GB (n<8) will be accessed in steps. RandFun sets these spaces as trap spaces. In practice, the size of application code is almost no more than 512GB, or even less than 1GB. When accessing the memory at the location n*1GB or n*512GB from the code segment, the side channel leakage will access the trap page. Then, we can find the probed function.

For Vector 2, the cloned process has the same address space as its parent process. Therefore, the child process inherits all trap spaces of the parent process. In addition, all code pages are still unreadable. As a result, RandFun can still capture various code probes in child processes. After that, the probed functions in the child process and its parent process will be marked as probed functions.

It should be noted that RandFun is only effective for code probes, not data probes. We leave data probes, such as the stack data disclosure based on *printf(%)*, to data layout randomization methods (such as stack ASLR). For code probes not mentioned in this paper, as long as they require reading code, accessing specific spaces or inevitably triggering specific signals, we can capture them. The essence of code probes is direct or indirect access to the memory where the code is located. We can dynamically manipulate the permissions and spaces of the code. As long as the behavior pattern of code probes is known or predictable, we can capture and filter out control flows with specific behavior characteristics, and then discover (potential) code probes.

B. Randomize the Probed Function

Existing ASLR methods suspend the target process during randomization, which causes runtime delay. Our protection target is the probed function, and the randomization granularity is the code blocks within the function. Directly manipulating the process's code while the process is running may cause a process crash. The component Randor (shown in the Fig. 1) is deigned to solve this problem. It copies the probed function into kernel space for randomization. A kernel thread *rand_thread* will be enabled to randomize the code copy. The thread and the process containing the probed function are parallel.

Compared with the existing methods, the location of the probed function itself will not be changed. The advantage of such a design is that the function can still be called normally by other callers. We don't need to search for and correct the callers calling the probed function. For complex closed source software, existing methods are difficult to construct accurate control flow graphs by analyzing binary code. Our design can precisely avoid this issue.

All ASLR methods must ensure that the randomized direct/indirect jump instructions can still reach the right locations. Direct jump refers to the code directly providing an offset relative to *rip* in the form of an immediate value in the instruction, and adding this offset to the *rip* pointing to the next instruction can obtain the target address of the branch instruction, such as

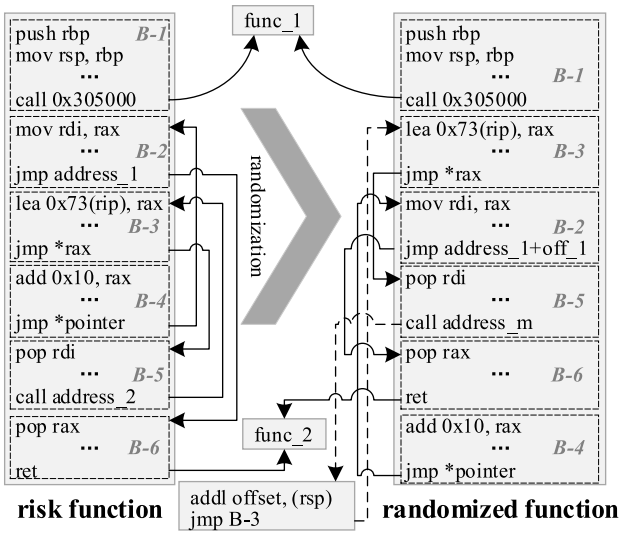

Fig. 3. The randomization of the risk function.

call/jmp offset. Indirect jump refers to the instruction's jump target coming from a register or pointer, which is an absolute address rather than an offset. The indirect jump instructions are also called ICT (indirect control flow transfer) instructions, such as *call/jmp *pointer/*register*.

1) Select the Code Blocks to be Randomized: For the probed function, we randomize it with the code block granularity. The code block selected by us containing at least one entry and one exit, and it can transfer control flows. The exit is a control flow transfer instruction. The next instruction adjacent to the exit is the entry of the next code block.

The exits selected by RandFun include *jmp offset*, *jmp *register/*pointer* (excluding PLT entries), *ret* and *call offset*, as shown in Fig. 3. The instructions need to be fixed in this paper are the direct jump instructions including *call offset*, *jmp offset* and all *jcc offset*(jump if condition is met) instructions. We can determine which instructions need to be fixed by analyzing the executable file of the target process.

2) Randomize Code Blocks: For direct jump instructions in the probed function, we recalculate their jump targets based on the new code layout and the original relative addresses after randomization. As a result, all such instructions can also reach the right locations. For *call offset*, its return path will be changed after randomization. Therefore, the return address on stack also needs to be modified. To achieve this goal, we modify the *offset* of *call offset* to redirect the control flow to a module, which can replace the return address and transfer the current control flow to a jump module after randomization. The jump module is in an unreadable page and it can transfer the return control flow to the real next instruction that have been randomized. This design not only enables the return control flow to be transferred to the right location, but also hides the return address on the stack from attackers. Therefore, once the function itself has been performed code probes, attackers cannot obtain the exact gadget addresses. Even the stack data can be stolen, they cannot obtain the real return addresses.

In addition, all addressing operations based on the current code address (stored in *rip*) also need to be fixed. Compared to direct jump instructions, rip-based addressing instructions are easier to recognize. The reason is that *rip* appears in such instructions, which can be obtained by analyzing the executable file of the process to be protected. For such instructions, we only need to correct the offset in the instruction.

The jump targets of *jmp *register/*pointer* are absolute addresses, which come from the parameters of the current function (such as the pointer of the callback function), or from the heap, stack, or data segment. The function parameter is determined before the probed function is called, and it will not be affected by the internal code block of the probed function. That is, the jump target determined by the parameter still points to the original address space and it doesn't need to be fixed. For the data in heap, stack, or data segment, it also does not need to be fixed. If the jump target has been written into memory before the probed function is called, it indicates that they are not controlled by the internal code blocks. If they are written after the probed function is migrated to the new space, it may be affected by the assigning statement. According to our observation, such data is accessed with the form of *rip+offset*, such as *lea 0x73(rip), rax*. Since the instruction is based on *rip*, the offset will be fixed as our design. We only need to modify the instructions writing data without manipulating the *jmp *register/*pointer*.

For *switch* and *try/catch* statements, they use the jump table (storing code block addresses) and *__gcc_except_table* (storing LSDA, language specific data area) to locate the jump targets. If the probed function uses *jmp *offset(rip)* to deference the control data in the jump table, or uses the function *_unwind_resume* to parse LSDA, the corresponding entries in the jump table or *_gcc_except_table* will be modified to point to the randomized code blocks.

3) Backfill the Randomized Function: The component CFbridge (in Fig. 1) is responsible for backfilling randomized functions into the original space. After randomization, the kernel thread *rand_thread* waits for a system trap. Then, CFbridge fills the randomized function back into the original space. Finally, the probed function in the new space is remapped to an non-executable page to capture the return control flows.

C. Protecting the Control Flow

1) Protect the Control Flow During Randomization: Due to the fact that the probed code is still executable during randomization, once adversaries immediately launch an attack after probing, the probed code snippets can be used as gadgets. To address this issue, the component CFer (shown in Fig. 1) captures all control flows jumping to the probed code and checks their legitimacy during the randomization.

To capture the control flows jumping to the probed function and ensure the probed function can be called legally, we create a new address space. The new space is the same size as the original address space, as shown in Fig. 4. In the original address space, the probed function will be mapped to new physical page(s), and the code of the probed function on that page(s) is *0xcc* (*int3*). As a result, calling the probed code in the original

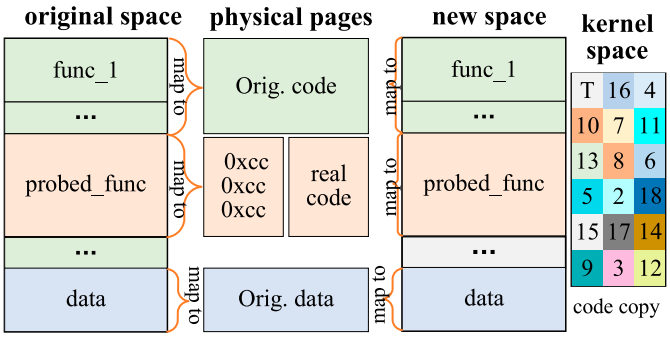

Fig. 4. The mapping method of the probed function.

space will trigger a system trap due to *int3*, which can be captured by RandFun. In the new space, the probed function is mapped to the original page(s), where the original function code is stored. To avoid legal control flows jumping back to the probed function in the original space, we scan the stack and modify the return addresses pointing to the probed function to point to the code in the new space (if necessary). By analyzing the Intel PT packet, we can determine whether there are return addresses to be modified on the stack.

However, if the probed function is located in a shared library, the above design will cause some problems. First, directly rewriting the shared code may cause conflicts. Second, the new space is only mapped to the address space of the current process instead of the address spaces of all processes. Therefore, the control flow of other processes cannot be transferred to the new space where stores the original code. To solve these problems, all the probed code pages that are shared will be copied to new physical pages. RandFun modifies the page tables of the current process to map the virtual address of the shared functions being probed to the new pages. So, the probed library code in new pages are private to the current process. Finally, RandFun can handle the private code pages without affecting other processes.

To ensure the probed function can still be legally called and its code blocks are not used as gadgets, all the control flows triggering system traps need to be checked. We propose a set of control flow detection strategies as follows:

1) If the instruction triggering a system trap is *call/jmp offset*, the control flow is legal. The reason is the *offset* is in an unwritable area and it cannot be tampered with. Moreover, the *offset* will be modified to point to the probed function in the new space. When it is called again, the system trap is no longer triggered, which reduces overhead.

2) The *call *xx* triggering a system trap must jump to the probed function's head in the original space.

3) The *jmp *xx* that triggers the system trap can only jump to the head of the opcode of an instruction.

4) If the instruction triggering the system trap is *ret* and it is in the new space, the control flow is illegal. In the new space, the return address paired with a *call* should point to the new space instead of the original space. When the adversary modifies the return address with the address of the probed code in the original space where the code has been rewritten with *int3*, a system trap will be triggered.

5) If the probed function and its caller are in different spaces (that is, they are in different mapped libraries, or the probed function is in a library and its caller is in an application), only *jmp *pointer* in PLT and *jmp *register* in dl_runtime_solve are allowed to transfer the control flow to the probed function, and the control flow must be transferred to the head of the function.

The above security strategies can quickly identify most but not all of the illegal *call/jmp *xx*. They need to be checked with stronger security strategies. *call/jmp *xx* (caller) is context sensitive. In context, the assignment and dereference to control data determine the destination (callee) of the control flow. Before the caller is executed, its control data is either explicitly assigned or implicitly assigned. In fact, CRAs hijack control flows by tampering with the assignments to control data or by changing the dereference to control data. We can judge the legitimacy of the control flow based on the changes in the caller context. The context detection strategies are as follows:

6) Explicit assignments. The explicit assignment means the original control data explicitly appears in code segment. For example, in "*lea 0x180 (rip), rax; ..., jmp *rax*", the control data comes from *lea 0x180(rip), rax*. Such control data is known and fixed. The basic reason that the control flow can be hijacked is the control data will be temporarily stored in the memory, such as stack, that can be tampered with. The explicitly assigned control data can be obtained by backtracking the historical paths recorded by Intel PT. In the attak scenario, the tampered pointer and the original pointer determined by the assignment statement are different.

7) Implicit assignments. The implicit assignment means that the original control data does not appear in code segment. Such data is determined by specific functions or compilers. For example, the VTables in C++ code segment is determined by the compiler.

For the control data stored in unwritable memory, such as jump tables in *.rodata*, it cannot be tampered with. An adversary can only tamper with the dereference to it. Such control data will be modified to point to the executable code in the new space instead of the probed code. If the control flow relying on the data triggers a system trap again, it indicates the dereference to the control data has been tampered with. The control flow is illegal. Moreover, the *vptr* should point to the head of the Vtable rather than its interior and the same *vptr* will not point to different Vtables at different time points.

For the control data that is implicitly assigned and stored in writable memory, such as the heap, it is copied from one location to another by memory copy functions/statements. Such control data will be modified to point to an unreadable transfer module that can transfer the control flow to the right code in the new space. At the same time, both the original control data and the ICT instruction that relies on the data will be recorded. If the ICT instruction triggers the system trap again, it indicates that the original control data has been changed. The control data should have the same directional characteristics before and after changing. For example, the function pointer should point to the function header instead of a code snippet inside the function before and after the changing. The data attribute will not be changed in the life cycle of the function. For example, the control data will not become non-control data, and the

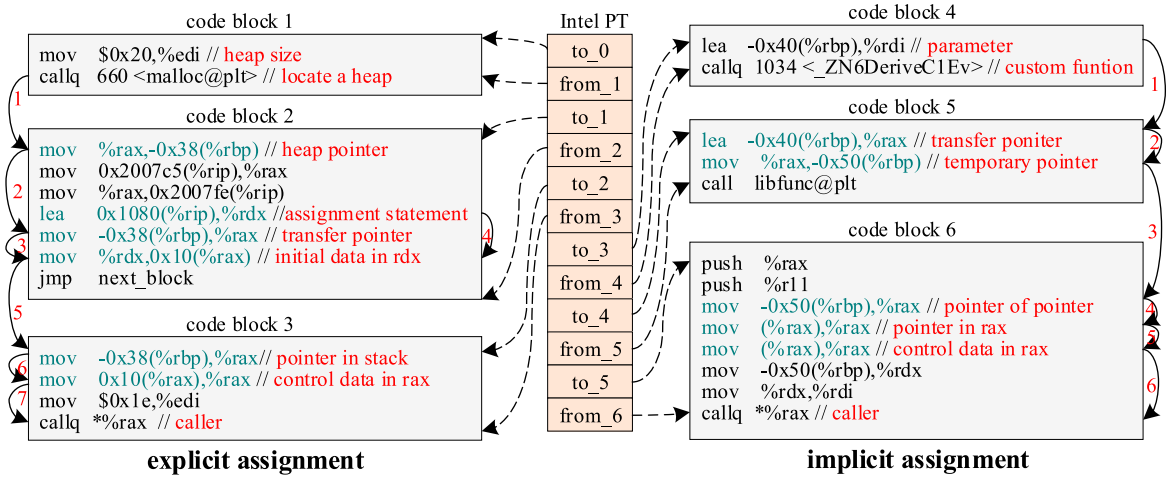

Fig. 5. The randomization of the risk function.

non-control data will not become control data. In addition, neither the ICT instruction nor its control data can violate strategies 2) $\sim$ 5).

8) Callback function. The callback function is a special case in which callers can correspond to multiple callees, which depends on the parameters passed to the caller. If the parameter is determined by an assigning statement, we use the strategy 6) to check the control flow. If the parameter is implicitly assigned, we use the strategy 7) to judge the legitimacy of the control flow.

To find the assigning statements or memory copy functions, we exploit Intel PT to record the instruction paths. The source of the original control data can be obtained from the paths, as shown in Fig. 5. Starting with the caller's operands, we trace backwards all the code blocks that have been executed in the current function until we get the final assigning statement. If no assigning statement to the control data is found in the current function, the control data is implicitly assigned. It is important to note that the callback functions use parameters as the caller's control data. If the operand is derived from a parameter of a function, the last executed function will be a retrospective target until an explicit assigning statement is found or an implicit assignment is identified.

For the control flows that have been identified as legal, we need to transfer them to right locations. After a system trap occurs, we modify the *guest rip* in VMCS to make it point to the executable code in new space. When the OS switches from *host* to *guest*, the control flow can be redirected to the right location.

2) Protect the Control Flow After Randomization: After the probed function is backfilled to the original space, the control flow will jump to an unknown code block if an adversary uses the original code address as a gadget address. As a result, it is almost impossible for the code block currently being executed to connect the next gadget. Therefore, RandFun can effectively prevent the CRAs using the probed code blocks as gadgets. So, we no longer need to track and analyze the control flow that jumps inside the randomized function in the original space. Nevertheless, CRAs (such as COOP) that use a full function as a gadget can still be deployed. Because RandFun only randomizes the code blocks in the function, and the function location is not changed. Such a function still has the risk of being used as a gadget if it has been known by adversaries. To prevent such an attack, we need to track and detect the control flows jumping to the header of the probed function after randomization.

We mark the header of the probed function with *int3*. As a result, all control flows jumping to the probed function header for the first time can be captured. We modify the *guest rip* in VMCS to redirect the control flow to a detection module. The module still uses the strategies proposed in section 5 to check the control flow. If the control flow is legal, it replaces the target of the caller instruction with the address of a transfer site. The transfer site can execute the header of the probed function and redirect the control flow to the right location through a jump instruction.

In theory, RandFun still has some limitations in terms of security, as it cannot handle all CRAs. If an attacker is able to deploy CRAs without code probes, RandFun can be bypassed. Deploying such attacks has strict limitations. Attackers must know the code layout and code forms of the closed-source software in advance, so that they can directly obtain available gadgets without code probes. Such a condition are clearly impossible in an OS with ASLR (especially fine-grained ASLR) enabled. In addition, if an attacker can hijack the direct jump branch by tampering with non-control variables, RunFun can also be bypassed. For example, for "*if (a) call func-1; else call func-2*", attackers can alter the control flow path by tampering with the variable *a* on the stack. Fortunately, the control flow hijacked by such an attack can only flow to a fixed location rather than to any locations requested by CRAs.

VI. Evaluation

We conduct all experiments on a Dell PC equipped with an i7-6700@3.4GHZ CPU and 16GB memory. The OS is Ubuntu-20.04 with kernel 5.1.4. All performance results are averaged after 10 runs.

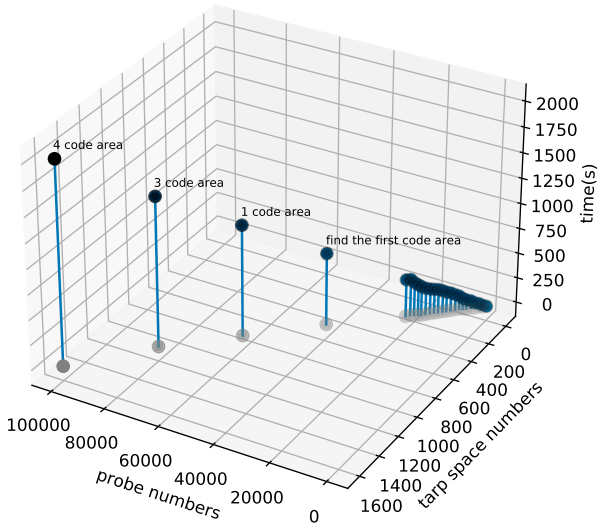

Fig. 6. Detect vector 1.

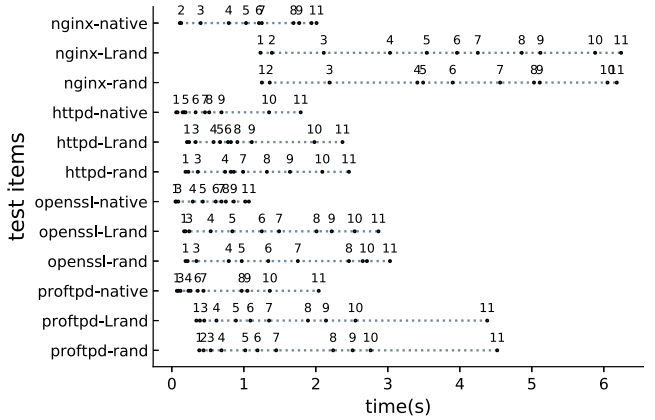

Fig. 7. Minimum time to obtain the turing-complete gadget set with a timeline for new gadget type leaks. Each (●) with a number n on top of it represents the time to leak/randomize n gadget types. xx-native: jitrop-native runs in the native OS; xx-RandFun: jitrop-native runs in the OS equipped with RandFun, xx-rand: RandFun randomize the function.

A. Security Evaluation

Inspired by IH [32], we test the code probe allocation oracle (Vector 1) [1] on Nginx in original OS. In our test, Vector 1 takes about 7ms-14ms for one probe. It usually needs 70 probes and 500ms-1000ms to probe one mapped code area. Under the protection of RandFun, Vector 1 still maintains a time of 7ms-14ms to get one mapped area. However, it takes about 10-15 minutes to get an area containing the real code page, as shown in Fig. 6. Because most of the mapped areas obtained by Vector 1 are trap spaces set by RandFun, which have been mapped to a trap page. Our test shows that one real code area is contained in tens of thousands trap spaces, while Vector 1 cannot identify which mapped area is the real code area. In general scenarios, Vector 1 is just an auxiliary tool and it cannot locate available gadgets. It will immediately notify the memory scanning tool to analyze the mapped area after locating it. From a probability perspective, the probability of the analyzed area being trap spaces exceeds 99.99%. When searching for gadgets with ROPgadgets in original OS, attackers can discover 20 gadgets within 300ms. However, under RandFun's protection, the ROPgadget will be detected when scanning the first binary code.

To deploy arbitrary jump (Vector 4), 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. 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. If the return address is tampered with a random 64-bit address, the probability of triggering SIGILL, SIGSEGV and EPT exception exceeds 99%. Then, RandFun can obtain the probed function. In addition, RandFun will trace upward according to the LBR and judge the legitimacy of the executed code blocks through the security strategies, until a legal code block is found.

For other probes including Vector 2, Vector 3, and Vector 5, they will be captured due to accessing trap pages, triggering exception signals, or causing EPT violation. Then, RandFun performs runtime randomization for the probed function.

RandFun's main idea is to change the address of the probed code before the gadget is executed. After a code probe occurs, RandFun randomizes the probed code. If we can complete randomization before the probed code is executed, CRAs cannot be deployed. In addition, detecting control flows that jump to the functions that have been probed but have not yet been randomized can also prevent CRAs. We use the modified jitrop-native [33] (the modified one can output more code addresses and more probing time) to probe gadget types in real applications. We measure the probe time and randomization time, which can verify the effectiveness of RandFun's runtime randomization. The results are shown in Fig. 7.

In the presence of RandFun, code probes become slower. The reason is jitrop-native frequently reads the process code, which causes system traps. We found that the runtime randomization is so fast that it can be completed in a short time after a code probe. More interestingly, some runtime randomization can even be completed before jitrop-native finds new gadget types. The reason is when jitrop-native reads the first byte of a function, the function will be randomized by a parallel kernel thread. Before the gadget in this function is found, the parallel kernel threads may have completed randomization. In a word, since multiple (up to twice the number of CPU cores) parallel kernel threads are used, RandFun can quickly randomize the probed code. As a result, before the attacker obtains enough gadgets, the previously probed gadgets have become unavaiable.

B. Performance Evaluation

SpecCPU2006 is used to measure the CPU overhead. In practice, the benign applications do not probe the code. Therefore, we cannot measure the impact caused by handling code probes. To solve this problem, we use an LKM (loadable kernel module) to read the code of the target function. It only reads one byte to trigger runtime randomization. The number of probed functions

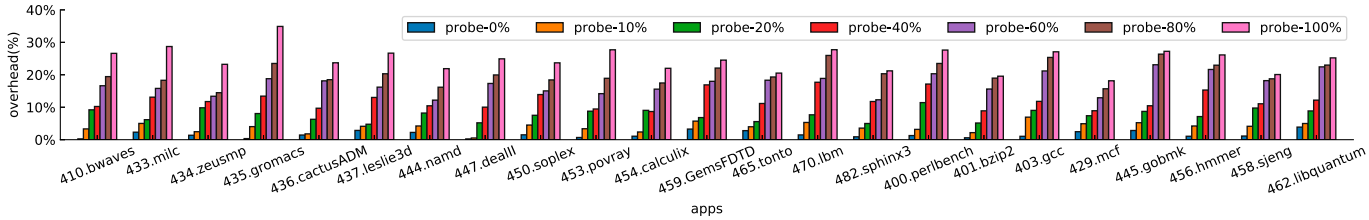

Fig. 8. The CPU overhead introduced by RandFun.

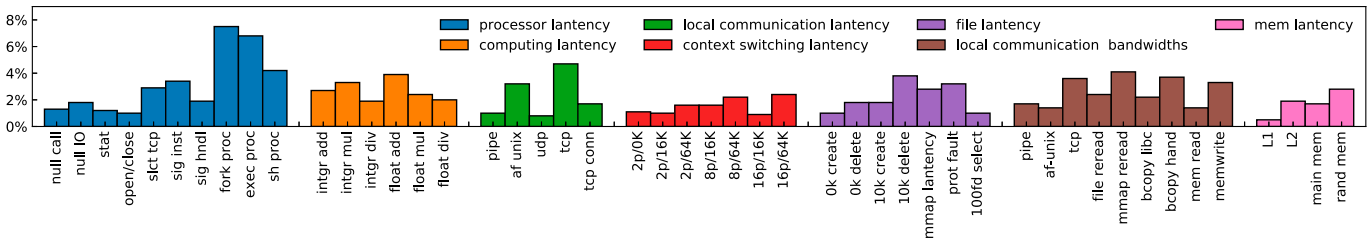

Fig. 9. The latency overhead introduced by RandFun.

increases from 0 to 100%, and all results are normalized based on original OS, as shown in Fig. 8.

In the scenarios without code probes, RandFun only introduces an average of 1.6% CPU overhead. When 10% functions in a process are probed, the average overhead is 3.9%. When 100% functions are probed, the average overhead is more than 30%. When a code probe occurs, RandFun needs to randomize the probed function and judge the legitimacy of the control flow. The more functions are probed, the greater impact on the running process. We believe that attackers should avoid performing large-scale and prolonged code probes on the target process. The reason is the scanning activities can slow down the target process and increase CPU/memory/network load, which will attract the attention of defenders. Therefore, we believe RandFun's overhead on the protected process is acceptable.

In real execution scenarios, code probes rarely occur. To observe the overhead introduced by RandFun in the scenarios without code probes, we use Lmbench to test the impact of RandFun on system latency and bandwidth, and the results are shown in Fig. 9. The average system latency is 2.4%, and the bandwidth overhead is 2.7%.

IOmeter is used to measure the impact of RandFun on I/O throughput and I/O response time when there is no code probe, as shown in Fig. 10. RandFun reduces the I/O throughput by 3% on average and increase the I/O response time by 3.9% on average.

Similar with BUDDY [36], we use Apache httpd to measure the overhead of RandFun on the network, as shown in Fig. 11. We measure the performance of httpd with different numbers of worker processes (p) and different numbers of concurrent connections (c). The size of requested file is 1MB. The average network delay of each group is 2.5%, 3.1%, 3.4%, 3.2%, 4.9% and 3.9% respectively.

The above measurements show that RandFun does not introduce excessive overhead to system latency, local bandwidth,

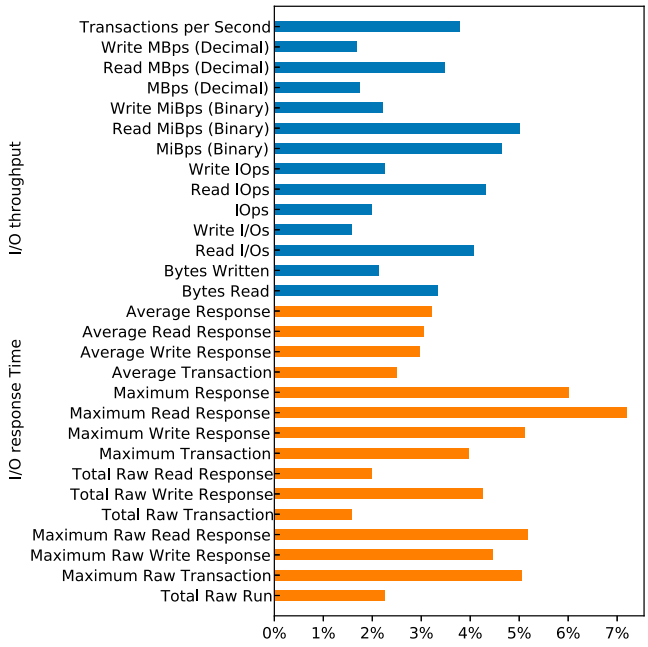

Fig. 10. The I/O overhead introduced by RandFun.

I/O, and network speed in normal scenarios. In fact, the overhead mainly comes from control flow detection and system traps. In the scenario without code probes, RandFun does not analyze the legitimacy of the control flow. However, the system trap events are inevitable. The system traps can be divided into unconditional traps and conditional traps. In the guest, the instructions *cpuid, gettsec, invd, xsetbv* and all VMX instructions except *vmfunc* will cause system traps unconditionally. Conditional traps are triggered by the specific events set by RandFun, which relates with code probes and control flow detection.

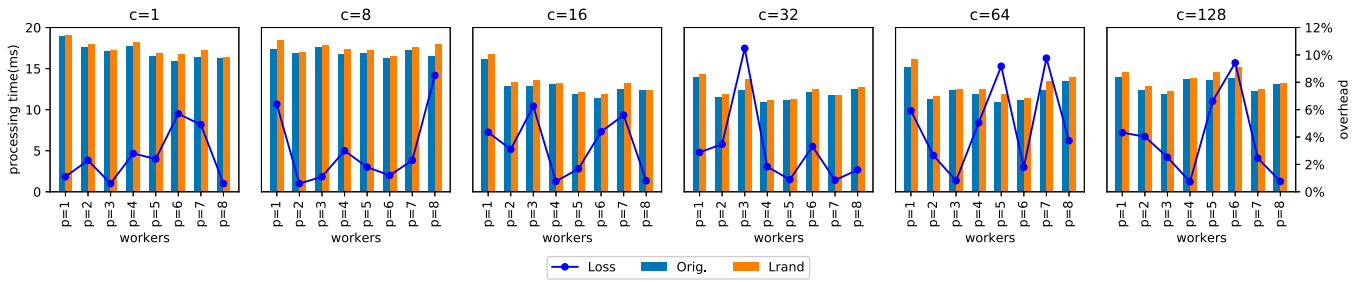

Fig. 11. The impact on network throughput.

TABLE I
MICRO BENCHMARKS

Native OS							OS With Lrand											
CA	JA	CP	JP	MinF	MajF	PW	ES	ST	CA^{1st}	CA^{2nd}	CP^{1st}	CP^{2nd}	JA^{1st}	JA^{2nd}	JP^{1st}	JP^{2nd}	MinF	MajF	PW
2.7	2.5	2.7	2.6	1321	$3.9*10^5$	16.5	96.4	499	1245	2.7	$1.3*10^6$	2.8	2.6	2.5	$1.1*10^6$	3.2	1334	$4.2*10^5$	81.4

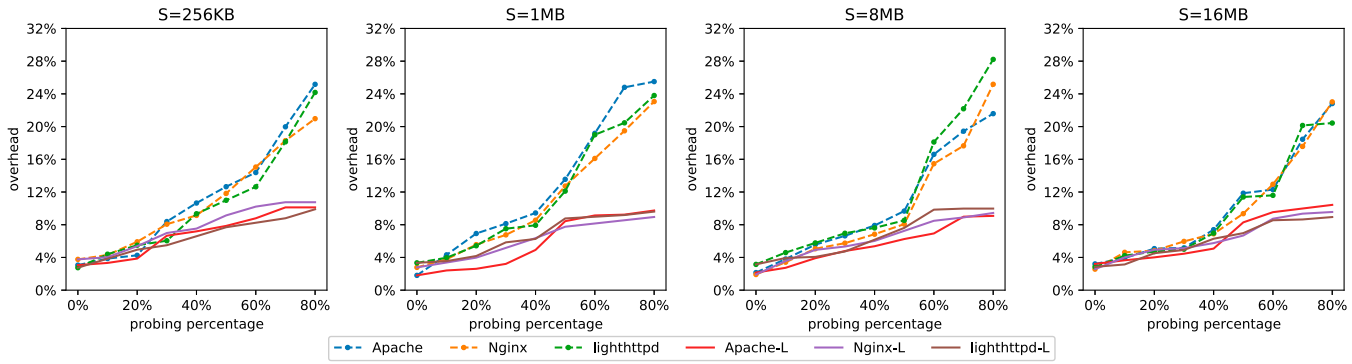

Fig. 12. Reduced overhead with process execution.

We use some micro benchmarks to measure the impact of RandFun on the randomized functions, as shown in Table I. The probed function will be filled back into the original space after randomization. After that, the control flow that jumps to the probed function head will trigger a system trap and it needs to be checked. Then, the control flow will be redirected to the transfer site. Therefore, the instructions that jump to the probed function head for the first time are time-consuming. For the legal instruction *call offset*, its operands will be modified with the *offset* of the transfer site when it is captured due to a system trap. Then, there are no system trap and control flow detection when the *call offset* is executed again. For the control data that is explicitly assigned, it will be fixed to point to the transfer site, which also avoids the system trap and control flow detection. Therefore, the $CP2^{nd}$ and $JP2^{nd}$ are faster. In addition, the speed of the page walk will be significantly affected. The reason is EPT leads to more page table walking.

In principle, the impact of RandFun on the probed process will gradually decrease with the process running. To verify this conclusion, we use web servers to measure RandFun, as shown in Fig. 12. For these web servers, the number of work processes is 4, the number of connections is 32, and the size of the requested file (S) is increasing. Before the web servers transfer data, we use a memory scanning tool to read the process code proportionally, which is a simulated code probe. After that, web servers are immediately used to transfer data. The overhead at this time is shown by the dotted line in the figure. Next, we continue to transfer data without measuring the overhead. 10 minutes later, we transfer the data again and measure the current overhead. The results are shown as solid lines. In contrast, the overhead introduced by RandFun gradually decreases. The reason is more and more legal control flows have been found with the process execution. RandFun corrects them so that the subsequent system traps and control flow detection become less and less.

To test the overhead in the scenarios where the CPU needs to be preempted, we run network applications with multiple work processes, as shown in Fig. 13. The CPU we use supports up to 8 threads. To ensure that the running processes can occupy all CPU cores, we set the minimum number of workers for the network applications to 8 and gradually increased it. At the same time, we continuously increased the code probe percentage. The size of the requested file is 1MB, and the number of connections is 32. We still respectively test the overhead at 0 and 10 minutes after the code probe occurs. The experiment results show that in the scenarios where all CPU cores have been occupied by the running processes, RandFun has a greater impact on the protected processes. The reason is that when code

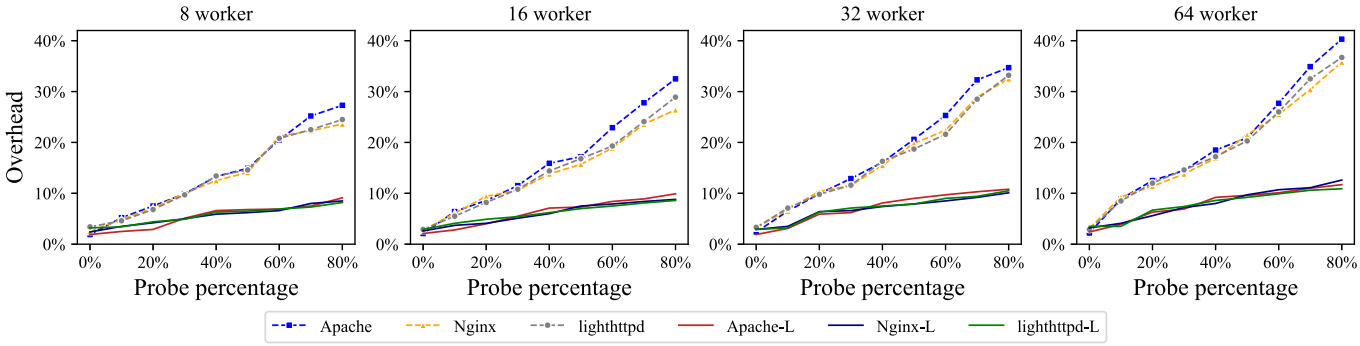

Fig. 13. The impact on multiple processes.

TABLE II
COMPARISON WITH THE EXISTING ASLR METHODS. AS: ACCESS TO SOURCE CODE; CM: COMPILER MODIFICATION; KM: KERNEL MODIFICATION; LiM: LINKER LODIFICATION; LoM: LOADER MODIFICATION; G: GRANULARITY; InP: INVALIDATION POINTS; RS: RANDOMIZATION SCOPE; U: USER-SPECIFIED; NUMBER X: EVERY XMS; SC: SYSTEM CALL; F: FORK(); P: PROBING EVENT; L: LOAD TIME; C: COMPILE TIME; F: FUNCTION; B: BASIC BLOCK; M: MODULES; S: THE WHOLE SEGMENT OR STACK/HEAP; O: OVERHEAD WITH SPEC OR OTHER BENCHMARKS; INS: INCREASE THE SIZE OF FILE OR CODE

	AS.	CM	KM	LiM	LoM	G.	InP	RS	O	Ins
RandFun (our method)	n	n	y	n	n	B	P	F	1.6%	1.1% $\sim$ 6%
Mixr [26]	n	n	n	n	n	U	U	U	1.66x	3.51x
Remix [21]	y	y	n	n	n	B	U	U	2.8%	14.8%
TASR [22]	y	y	y	y	y	S	SC	S	2.1%	1MB
CodeArmor [20]	n	n	n	n	y	S	SC	S	6.9%	4.4% or 13.4%
STABILIZER [8]	y	y	n	n	n	F	500	S	6.7%	-
Chronomorph [34]	y	n	n	n	n	B	U	S	-	$>$ 50%
Shuffler [9]	y	n	n	n	y	B	50	S	14.9%	74% $\sim$ 117%
Reranz [31]	n	n	y	n	n	B	SC	S	6%	73MB
CCR [25]	y	y	n	y	n	B	L	S	0.28%	11.46%
SafeHidden [19]	n	n	n	n	n	S	f/SC/P	U	2.75%	-
Adelie [23]	y	y	n	n	n	B	C	S	$<$ 2%	-
RuntimeASLR [7]	n	n	n	n	n	M	f	S	$>$ 217x	-

probes occur, RandFun needs to preempt the CPU. If there are multiple processes being probed, RandFun needs to create more threads for randomization, which further increases the number of threads that preempt the CPU. For the running processes, the more processes being probed, the greater the CPU load. This results in their CPU time slices being frequently preempted by RundFun, which affects their running speed. Fortunately, the overhead caused by RandFun on protected processes does not last long, and it will gradually decrease. The reason is that the threads used for randomization will exit after completing the randomization. Moreover, more and more legal paths will be identified as process execution, and less and less control flows need to be detected.

Although the trap spaces are huge, they do not occupy too many page tables. Because most page tables at all levels are shared. For example, *0x3f298000ff18* and *0x4a501000e297* use different entries in the first level page directory, while the addresses in their entries point to the same second page table; All entries in the second level page table point to the same third level page table, all entries in the third level page directory point to the same fourth level page table, and all entries in the fourth level page table point to the same trap page. According to our observation, the memory occupied by all trap spaces does not exceed 64KB. If the probed target is shared library, all probed code will become private to the current process. Therefore, the more shared code being probed, the larger the memory overhead.

C. Comparison and Discussion

Similar to Mixr [26], we compared RandFun with the current ASLR solutions, as shown in Table II. The results show the overhead of RandFun is smaller. Compared with other methods, RandFun only randomizes the probed function instead of the entire code segment, and the randomization is parallel to the process. Therefore, the randomization target is smaller and the process suspending time is shorter, which reduces the randomization complexity and the delay. To the best of our knowledge, RandFun is the first runtime randomization method that only handles the probed functions instead of the entire code segment.

Moreover, we also compared RandFun with the existing CFI (control flow integrity) methods, and the results are shown in Fig. 14. Except for RP, all other indicators are the qualitative

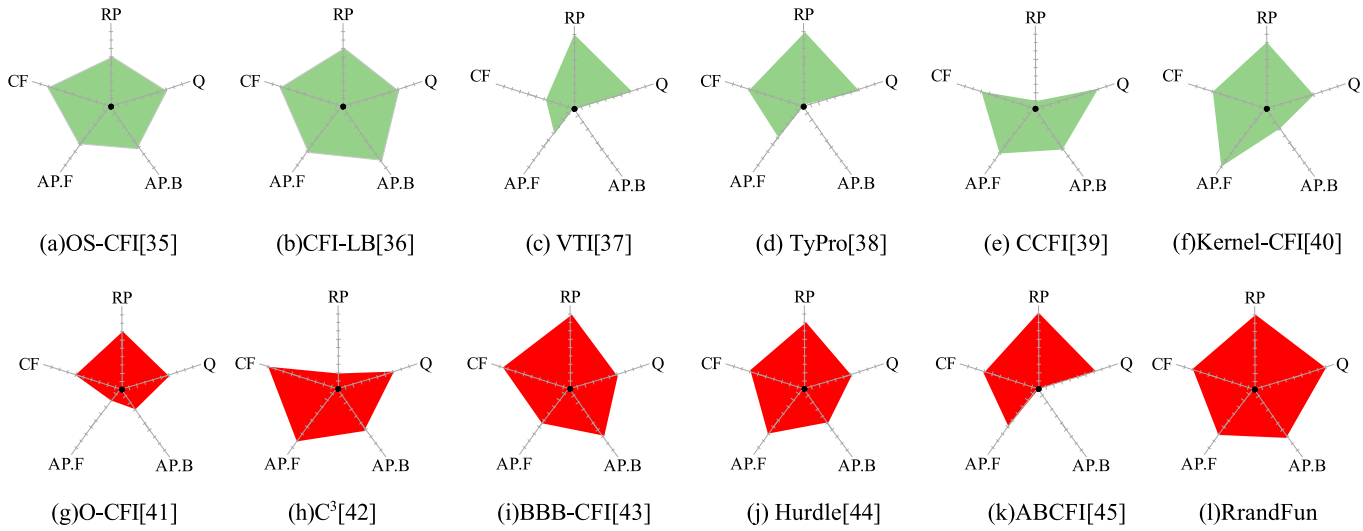

Fig. 14. Comparison with other methods.

result that analyzes the security method's defense principles, whose definitions are shown as (1)∼(4).

$$CF = \sum_{k=1}^{N} (P_{app}^{k} \times I_{app}^{k} + P_{lib}^{k} \times I_{lib}^{k}) \tag{1}$$

$$AP.F = \sum_{m=1}^{M} P_{attack}^{m} \times \frac{F\_I_{attack}^{m}}{F\_I_{all}^{m}} \tag{2}$$

$$AP.B = \sum_{m=1}^{M} P_{attack}^{m} \times \frac{B\_I_{attack}^{m}}{B\_I_{all}^{m}} \tag{3}$$

$$Q = \sum_{n=1}^{tar\_num} P_{mal}^{n} \times \frac{1}{L_n} \tag{4}$$

P_{app}^{k} and P_{lib}^{k} respectively represent the probability that the k^{th} control flow transfer instruction (including *call *pointer/*register*, *ret* and some other instructions that can be used as gadgets) can be tracked in the application and library. I_{app}^{k} and I_{lib}^{k} respectively represent the percentage of the k^{th} control flow transfer instructions in applications and libraries. The number of the control flow transfer instructions to be tracked is positively correlated with CF. If all the instructions that can be used as gadgets by CRAs can be tracked, an ideal CF can be obtained. P_{attack}^{m} represents the probability that the m^{th} attack scenario can be identified. $F\_I_{attack}^{m}$ and $B\_I_{attack}^{m}$ respectively represent the number of the illegal forward instructions and the number of the illegal backward instructions in the m^{th} attack scenario. $F\_I_{all}^{m}$ and $B\_I_{all}^{m}$ respectively represent the number of all tracked forward instructions and the number of all tracked backward instructions in the m^{th} attack scenario. In a normal scenario, all control flows are legal. At this time, it is unnecessary to track and detect the legal control flows. Screening out the potential attack scenarios can reduce the unnecessary tracking and detection, which is the key to improving the efficiency of security methods. Another word, only tracing the control flow transfer instructions in the attack scenario can obtain the high AP.F and AP.B.

The factors affecting Q mainly include three aspects: attack principles, defense principles and the characteristics of the attack objects. We qualitatively analyze Q around the three aspects. *Tar_num* represents the total number of control flow transfer instructions. P_{mal}^{n} represents the probability of the n^{th} control flow transfer instructions that can be maliciously used. It is determined by the characteristics of the attack principles and the characteristics of the attack objects. L_n represents the jump target number of the n^{th} control flow transfer instruction in the attack. In practice, each control flow transfer instruction has only one legal jump target when it is executed. If a security method can ensure that each control flow can be transferred to the only one legal target, it can get an ideal Q. That is, the ideal L_n is 1. The larger the L_n, the smaller the Q. For the code that cannot be used by an attacker, its P_{mal}^{n} is 0. Therefore, it does not have any effect on improving Q, which is reasonable to profile Q. The larger the P_{mal}^{n}, the greater the risk of the n^{th} instructions. So, the more a security method protects such instructions, the more its effort contributes to improving Q.

In summary, RandFun is a more balanced method. Compare with the existing methods, it randomizes the specific function instead of the entire code segment only when the probed function is detected. Therefore, RandFun has stronger target specificity and scenario specificity. The randomization triggered by code probes can reduce unnecessary randomization. Runtime randomization in parallel with the process can reduce the suspending time of the process, thereby reducing the running delay. Moreover, the partial randomization can reduce the size of the protected object, thereby reducing the randomization complexity. In addition, RandFun can also protect control flow during randomization and mitigate the COOP attack after randomization. We believe that the event-triggered runtime partial randomization is a very promising security method.

VII. Conclusion

This paper proposes a runtime partial randomization method RandFun. It can perceive code probes during process running. Unlike existing methods, we do not directly block the potential code probes, but rather randomize the probed functions and detect control flows that jump into them. The probe activity is used as a trigger event for randomization, and the probed function is the only target to be randomized. In our design, the randomization and process execution are parallel. To ensure the probed function can be called normally during randomization, the function will be migrated to a new space. In addition, all control flows that jump into the probed function are detected. Compared to existing methods, RandFun has stronger target specificity and scenario specificity. Experiments and analysis show that RandFun has a good defense effect against code probes and CRAs, and only introduces 1.6% CPU overhead.

References

[1] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur. 16)*, 2016, pp. 121–138.

[2] W.-L. Mow, S.-K. Huang, and H.-C. Hsiao, "LAEG: Leak-based AEG using dynamic binary analysis to defeat ASLR," in *Proc. IEEE Conf. Dependable Secure Comput. (DSC),* Piscataway, NJ, USA: IEEE Press, 2022, pp. 1–8.

[3] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 574–588.

[4] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proc. IEEE Symp. Secur. Privacy,* Piscataway, NJ, USA: IEEE Press, 2016, pp. 969–986.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières and D. Boneh, "Hacking blind," in *Proc. IEEE Symp. Secur. Privacy,* Piscataway, NJ, USA: IEEE Press, 2016, pp. 227–242.

[6] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Proc. Netw. Distrib. Syst. Symp.*, vol. 17, 2017, p. 26.

[7] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make ASLR win the clone wars: Runtime re-randomization," in *Proc. Netw. Distrib. Syst. Symp.*, 2016, pp. 1675–1689.

[8] C. Curtsinger and E. Berger, "Stabilizer: Statistically sound performance evaluation," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 219–228, 2013.

[9] D. Williams-King et al., "Shuffler: Fast and deployable continuous code re-randomization," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 367–382.

[10] A. Milburn, E. Van Der Kouwe, and C. Giuffrida, "Mitigating information leakage vulnerabilities with type-based data isolation," in *Proc. IEEE Symp. Secur. Privacy*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 1049–1065.

[11] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 437–452.

[12] S. Carr and M. Payer, "DataShield: Configurable data confidentiality and integrity," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 193–204.

[13] A. Brahmakshatriya and P. Kedia, "ConfLLVM: A compiler for enforcing data confidentiality in low-level code," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–15.

[14] M. Neugschwandtner and A. Sorniotti, "Memory categorization: Separating attacker-controlled data," in *Proc. Detection Intrusions Malware, Vulnerability Assessment: 16th Int. Conf.*, 2019, pp. 263–287.

[15] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1607–1619.

[16] E. Van Der Kouwe, T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida, "Type-after-type: Practical and complete type-safe memory reuse," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 17–27.

[17] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz, "CoDaRR: Continuous data space randomization against data-only attacks," in *Proc. 15th ACM Asia Conf. Comput. Commun. Secur.*, 2020, pp. 494–505.

[18] L. Davi, D. Gens, C. Liebchen, and A. R. Sadeghi, "PT-Rand: Practical mitigation of data-only attacks against page tables," in *Proc. Netw. Distrib. Syst. Symp.*, 2017, pp. 1–15.

[19] Z. Wang et al., "SafeHidden: An efficient and secure information hiding technique using re-randomization," in *Proc. USENIX Secur.*, 2019, pp. 1239–1256.

[20] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the code space to counter disclosure attacks," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, 2017, pp. 514–529.

[21] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, 2016, pp. 50–61.

[22] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 268–279.

[23] R. Nikolaev, H. Nadeem, C. Stone, B. Ravindran, "Adelie: Continuous address space layout re-randomization for Linux drivers," in *Proc. ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2022, pp. 483–498.

[24] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 280–291.

[25] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis and M. Polychronakis, "Compiler-assisted code randomization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2018, pp. 461–477.

[26] W. Hawkins, A. Nguyen-Tuong, J. D. Hiser, M. Co, and J. W. Davidson, "Mixr: Flexible runtime rerandomization for binaries," in *Proc. Workshop Moving Target Defense*, 2017, pp. 27–37.

[27] K. Lu, M. Xu, C. Song, T. Kim and W. Lee, "Stopping memory disclosures via diversification and replicated execution," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 1, pp. 160–173, Jan./Feb. 2021.

[28] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 745–762.

[29] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proc. 5th ACM Conf. Data Appl. Secur. Privacy*, 2015, pp. 325–336.

[30] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2015, pp. 605–622.

[31] Z. Wang et al., "ReRanz: A light-weight virtual machine to mitigate memory disclosure attacks," in *Proc. 13th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2017, pp. 143–156.

[32] Z. Wang et al., "Making information hiding effective again," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2576–2594, Jul./Aug. 2022.

[33] S. Ahmed, Y. Xiao, K. Z. Snow, G. Tan, F. Monrose, and D. Yao, "Methodologies for quantifying (Re-) randomization security and timing under JIT-ROP," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 1803–1820.

[34] S. Friedman, D. J. Musliner, and P. K. Keller, "Chronomorphic programs: Runtime diversity prevents exploits and reconnaissance," in *Proc. Int. J. Adv. Security*, 2015, pp. 120–192.

[35] M. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *Proc. USENIX Secur. Symp.*, 2019, pp. 195–211.

[36] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 95–110.

[37] D. Bounov, R. G. Kici, and S. Lerner, "Protecting C++ dynamic dispatch through VTable interleaving," in *Proc. Netw. Distrib. System Symp.*, 2016, pp. 1–15.

[38] M. Bauer, I. Grishchenko, and C. Rossow, "TyPro: Forward CFI for C-style indirect function calls using type propagation," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, 2022, pp. 346–360.

[39] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 941–951.
[40] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, 2016, pp. 179–194.
[41] Y. Lin, X. Cheng, and D. Gao, "Control-flow carrying code," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2019, pp. 3–14.
[42] W. He, S. Das, W. Zhang, and Y. Liu, "BBB-CFI: Lightweight CFI approach against code-reuse attacks using basic block information," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 1, pp. 1–22, 2020.
[43] C. DeLozier, K. Lakshminarayanan, G. Pokam, and J. Devietti, "Hurdle: Securing jump instructions against code reuse attacks," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 653–666.
[44] J. Li, L. Chen, G. Shi, K. Chen, and D. Meng, "ABCFI: Fast and lightweight fine-grained hardware-assisted control-flow integrity," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3165–3176, Nov. 2020.

YongGang Li received the Ph.D. degree from the University of Science and Technology of China, in 2019. He was a Postdoctoral Fellow with the Chinese University of Hong Kong, Shenzhen. Currently, he is an Associate Professor with the School of Computer Science and Technology, China University of Mining and Technology. His research interests include computer architecture, virtualization principle, cloud computing, and system security.

Yu Bao received the Ph.D. degree from Tongji University, in 2011. Currently, he is a Staff Engineer with the Security Department of Computer Science and Information Technology Institute, China University of Mining and Technology. His research interests include information security and privacy in AI distributed network and cyber security in IoT.

Yeh-Ching Chung (Senior Member, IEEE) received the Ph.D. degree in computer and information science from Syracuse University, in 1992. Currently, he is a Professor with the Chinese University of Hong Kong (CUHK), Shenzhen. His research interests include parallel and distributed processing and system software.