RESEARCH

DID: Data Isolation via Dynamic Adjustment of Permissions and Spaces

YongGang Li[1], Zhi Qu[1*] and Yeh-Ching Chung[2]

Abstract

Heap and stack are the main carriers for storing data in the operating system (OS), and their isolation is a key factor to ensure the data privacy and instruction orders. The existing isolation methods have the problem of coarse granularity. In addition, the completely open feature of the read and write permissions of the heap and stack has a serious impact on isolation. To solve these problems, this paper proposes an isolation method DID. This method refines the granularity of the isolated stack and heap. It establishes a framework to dynamically adjust the permissions and spaces of the target data. Based on this framework, DID builds four mechanisms: heap fence mechanism, heap owner mechanism, stack alternation mechanism, and stack jump mechanism. These mechanisms can be used to isolate the data on the heap or stack. Experiments and analysis show that DID has a good defense effect against various attacks based on data out-of-bounds access. And it only introduces 1.9% overhead to the CPU.

Keywords: Access controls; Data Integrity; System Architectures

1 Introduction

The heap and stack in the OS are the main carriers for storing data. The data on the carriers should be isolated to ensure the data privacy and instruction orders. However, neither the heap nor the stack can provide strong isolation to the data. Therefore, the data on the heap and stack has become the preferred target of malware.

The overflow attack[1, 2, 3] is a main form to break data isolation. After the isolation is broken, the data stored on heap or stack will be overwritten. Attackers can use the tampered data to achieve malicious purposes such as denial of service[4], information theft[5, 6], and control flow hijacking[7, 8]. The vast majority of overflow attacks originate from code vulnerabilities, which can be used by code reuse attacks (CRAs)[9, 10, 11].

The essence of an overflow attack is the execution entity's out-of-bounds access[12, 13], between different objects. For the data on the heap and stack, the current OS only implements basic isolation between different execution entities through different address spaces. For the data between different objects (such as two heaps) in the same address space, the OS does not set strict isolation for them. Therefore, the execution entity cannot be detected and prevented when an out-of-bounds access occurs.

Building a data isolation mechanism for different objects is the key to solving these problems. However, the spaces of stack and heap, and the data on them are changed dynamically, which causes the problem of defining boundaries. For the data on the heap and stack, there are various forms of out-of-bounds access. Attackers can use different overflow vulnerabilities to tamper with control data or sensitive data. Then, illegal data will spread the taint information to the instruction flow or data flow. The out-of-bounds access does not cause an execution exception, which makes the taint information undetectable.

2 Related Works

To protect the data on the heap and stack, researchers have carried out extensive researches. The most popular methods include boundary detection, address space location randomization (ASLR) and memory isolation. We will introduce them separately.

2.1 Boundary detection

The boundary detection method sets flags at the tail of the heap and the stack, and determines whether the overflow attack occurs by checking the flags. SafeStack(+)[14] can diagnose and patch stack-based

*Correspondence: 04211302@cumt.edu.cn
[1]School of Computer Science and Technology, China University of Mining and Technology, XuZhou
Full list of author information is available at the end of the article

buffer overflow vulnerabilities automatically. It locates data flow sensitive variables on the unsafe stack that could potentially affect evaluation of branching conditions, and adds canaries of random sizes and values to them to detect malicious overwriting. But when it detects an attack, the application has already been attacked once, which may lead to unpredictable consequences. StackGuard[15], CanaryExp[16] and Libsafe[17] add a piece of detection data to the tail of the heap or stack to ensure that the target object cannot be accessed beyond its boundary. However, attackers can use memory disclosure (such as cloning a child process) to read the boundary value and use equivalent rewriting to bypass these methods.[18, 19, 20]

2.2 ASLR

Address space location randomization (ASLR) makes it difficult to gain accurate locations of data and code. HARM[21] performs non-bypassable randomization to the firmware running in a sandbox in the normal world. Compared with HARM, fASLR[22] loads a function from the flash and randomizes its base address in a randomization region in RAM when the function is being called. Code randomization only increases the difficulty of cross-boundary access, and cannot solve the problem of data isolation fundamentally. For example, LAEG[23] can efficiently recover base addresses from uninitialized buffers and use them to construct an exploit that is resilient to ASLR. In addition, the Clone-ROP[24] can also get the addresses through process cloning. To solve the clone problem, researchers perform periodic or real-time ASLR and AntiRead Method[25]. Although RUNTIMEASLR[24] and STABILIZER[26] have made improvements to defend against process cloning, they have no defensive effect on the attacks based on memory leakage (such as brute force cracking).

2.3 Memory isolation

Memory isolation isolates sensitive data from non-sensitive data, and only the specific objects can access sensitive data. The isolation methods include hardware-based methods and software-based methods.

Hardware-based methods. ERIM[27] and LibMpk[28]are hardware-based methods that uses Intel MPK to protect the sensitive data, and the code in the right domain can access the sensitive data. SeCage[29] and xMP[30] protect the target data via EPT (Extended Page Tables). SSF[31] generalizes the process and library and user/kernel isolation inside the TEE while allowing for efficient memory sharing. Generally, hardware-based methods require the size and position of protected targets to be relatively fixed. Facing dynamic objects, their protection effect and running efficiency will be significantly affected.

Software-based methods. Datashield[32] separates the memory into a precisely protected region for sensitive data and a coarsely protected region for non-sensitive data. ConfLLVM[33] is a compiler-based method, and it classifies the data into public data and private data. TDI[34] is also a compiler-based method, and it isolates memory objects with different colors in separate memory areas and uses compiler instrumentation to constrain loads to the area. Different with ConfLLVM[33] and TDI[34], MemCat[35] identifies the data that can be controlled by adversaries using compile-time policy, and it allocates the specific objects on a separate heap/stack. Type-After-Type splits available memory into separate pools for each type of heap data[36]. Overall, software-based methods require prior knowledge of which data is sensitive, which is not applicable to closed source software.

3 Over Architecture of DID

The OS divides the address space into user space and kernel space according to different permissions. The entire space is divided into multiple independent spaces with the granularity of the process. Such a coarse-grained isolation method makes it difficult to establish an isolation layer between objects in a same process. For example, the functions can access all the heap data in the same address space, even if the data does not belong to them. Moreover, the permissions of the data on heap and stack do not have any distinction for all control flows in the same address space. And the OS has no legitimacy detection for data access to the heap and stack. Therefore, the control flow's out-of-bounds access to these data does not cause any exceptions, which poses a huge challenge for abnormal behavior detection.

For heap data and stack data, enhancing the isolation between different objects (such as different function stack frames) can resist a variety of malicious behaviors such as overflow attack, code reuse attack (CRA), and memory leakage. To achieve this goal, two key problems need to be solved: one is the coarse-grained problem of the isolated object, and the other is the problem of their fixed permissions. DID establishes isolation mechanisms in units of heap and function stack frames to reduce the granularity of target objects. At the same time, the concept of the dangerous function was introduced to approximate the real attack scenario. In the specific scenarios, DID dynamically manages the read and write permissions of the target data to make it selective for control flow. That is, only the specific control flow can access the target data. In addition, DID also sets traps for the target objects to capture out-of-bounds access. We need to adopt different isolation methods for different execution entities.

In practice, the source code of the execution entities may not be available. We need to adopt different isolation methods for different execution entities. The overall architecture of DID is shown in Fig. 1. It consists of three components: DisAD, SouDis and RunDis.

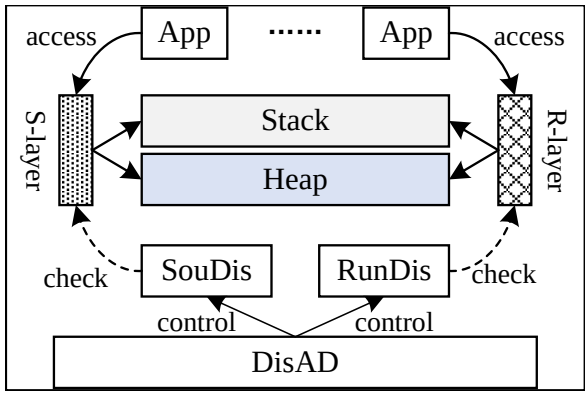

Figure 1 The overall architecture of DID

DisAD is used to dynamically adjust the space and permission of the target object, so that it can prevent the out-of-bounds access. Based on the adjustment ability provided by DisAD, SouDis and RunDis establish access verification layers S-layer and R-layer. The former is used to isolate the data on the target objects (stack and heap) with source code, while the latter is used to isolate the data on the target objects without source code. In fact, S-layer and R-layer are two functional abstractions. They can dynamically adjust the read and write permissions of heap and stack, the space range of data, and the execution permissions of code, thereby achieving data isolation.

In fact, not all the data need to be isolated. In most scenarios, control flow cannot break the isolation between different objects due to the legal execution logic. If and only if the legal execution logic is broken, an out-of-bounds access occurs. Heap overflow and stack overflow are the basic methods to break the legal execution logic. For example, the return address can be rewritten through a stack overflow function.

For the most out-of-bounds accesses, they are triggered by the functions with overflow risks, and these functions are called dangerous functions. The dangerous functions identified in this paper are shown in Table 1. In practice, only the dangerous functions containing variable parameters can cause overflow. Only the data in the scenario where such functions are being called need to be protected. All these functions are called via PLT (Procedure Linkage Table). Therefore, we can detect whether they are called by analyzing PLT. After that, we add a hook to the corresponding PLT entry to check whether the parameter of the function is a variable.

Table 1 The list of dangerous functions.

Header File	Functions
<string.h>	strcpy(), strncpy(), strccpy(), strcat() strdup(), memcpy(), bcpy(), getchar()
<stdio.h>	scanf(), sprintf(), snprintf(), fprintf() vsprintf(), sscanf(), fscanf(), gets()
<libgen.h>	strcadd(), strecpy(), strtrns()
<stdlib.h>	realpath()
<conio.h>	getch()

In the scenario where the source code is available, SouDis rewrites the function accessing the target heap in the source code. In the scenario where the source code is non-available, RunDis detects and monitors all contiguous heap owned by the execution entity dynamically. In addition, RunDis also distinguishes whether the control flow belongs to application code or library code.

To clarify the setting and function of our mechanisms, we describe them in the manner described in this paper[37]. The manner is shown as (1), and the symbol definition is shown as table 2.

$$\frac{Conditions\ Settings}{Operations\ and\ Execution\ Results} \tag{1}$$

Table 2 The symbols and their meaning.

Symbol	Meaning
$A := \{a, b\}$	Tuple A contains metadata a, b.
$a\|b$	Metadata a has attribute b.
$m \to n$	Replace the value or expression m with n.
$a \not\prec A$	The operation a occurs in tuple A.
$V \mapsto S$	Each metadata of tuple V has and only has one metadata corresponds to it in S, not vice-versa.
$a \propto M$	The object a belongs to memory M.
$I \triangleright C$	Instruction I is executed at the start of code C.
$V \bowtie S$	The metadata in the tuple S corresponds to the metadata in the tuple V one to one.
$f(a) :: \{e\}$	The function f operates on a with the expression e.

4 Design and Implementation of DisAD

To capture specific events related to data management, DisAD uses Intel VMX (Virtual Machine Extension) Non-Root and VMX Root to divide the native OS into two modes: guest and host. The formal definition of the new running modes is as model 1.

1. $Mod := \{guest, host\}$ // running modes of OS
2. $Eve := \{events | set\ through\ Vmcs\}$ // event setting
3. $\forall event \in Eve$ and $RunMod == Mod.guest$, // event occurs
4. $ModSwitch[guest \to host]$ // system trap

Model 1. New running rules

Under normal cases, the OS runs in guest. When the OS executes a specific event, it will fall into host from

guest (lines 3-4), which is called system trap. DisAD can set the system trap events by manipulating VMCS (Virtual Machine Control Structure) fields. DisAD is essentially a lightweight hypervisor, which can manage resource accesses in guest, such as changing memory permissions, modifying instruction paths, capturing specific events, and monitoring OS status, etc.

Based on the new running modes, DisAD can adjust the permissions and spaces of heap and stack dynamically. The protected heap and stack will show different characteristics due to the different control flows. To achieve this goal, DisAD needs to have the capability, the dynamic adjustment for memory permissions and object state.

DisAD uses the permission control bits (bits 0-2) of the EPT last-level page table to control the read, write and execute permissions of the memory. For a small amount of memory, we can use this method to control their permissions. However, if the permissions of a large amount of memory need to be adjusted dynamically, this method will cause frequent system traps, which introduces an obvious overhead. At the same time, this method also affects other execution entities when it manipulates the shared library code. To solve this problem, DisAD designs two memory access control methods, page table redirection and EPT redirection. The page table redirection method is shown in Fig. 2. We define the redirection rules as model 2.

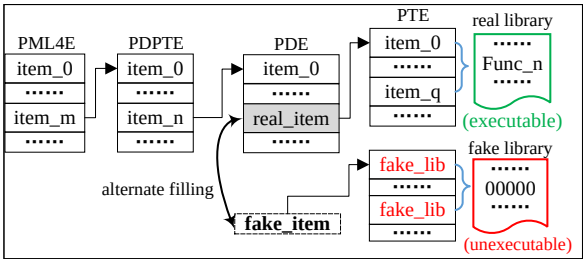

Figure 2 Dynamic adjustment for memory permissions

1. $real\_library := \{lib_1, lib_2, \ldots, lib_n\}$, $custom\_code := \{c_u\}$
2. $fake\_library := \{lib\_page | non - executable\}$
3. $MemRan(lib_k) = real\_item$, $FakeRan(lib_k) = fake\_item$

4. $\forall ins\_access \in fake\_library$ and $RunMod == Mod.guest$,
5. $ModSwitch[guest \rightarrow host]$,
6. $switch\_item[fake\_item \rightarrow real\_item]$, // change item
7. $ModSwitch[host \rightarrow guest]$,
8. $\forall ins\_jump[real\_library \rightarrow custom\_code]$, // jump
9. $ModSwitch[guest \rightarrow host]$,
10. $switch\_item[real\_item \rightarrow fake\_item]$,
11. $ModSwitch[host \rightarrow guest]$.

Model 2. Redirection rules

In this method, DisAD first calculates the size of the specified library (lib_k in line 1) and finds the page table entry real_item that can completely address the library (line 3). After that, DisAD creates a physical page for each level of page table after the level pointed to by real_item. Among them, the first address of the first created physical page is fake_item. All the entries in each new created physical page point to the next-level new physical page. The new physical page corresponding to the last-level page table is filled with the first address (fake_lib) of the virtual library fake_library. Fake_library is essentially a non-executable physical page (line 2). DisAD can control the execution permission of libraries by filling real_item and fake_lib alternately (lines 4-11), and does not affect other execution entities to call library functions.

The EPT redirection method can be divided into partial redirection and complete redirection according to the granularity of the targets need to be redirected. Partial redirection can be achieved by modifying the page table entry of the EPT to redirect it to a specified physical page. Complete redirection is implemented by the instruction vmfunc provided by Intel VMX. DisAD can redirect all physical pages of the process to any location or limit them to a fixed range without causing any system trap through the vmfunc.

The original page table redirection and EPT partial redirection methods do not need to inject code into the target object, so they are suitable for the situations without source code. However, the two methods need to be triggered by an EPT violation, so the OS will trap into host. EPT complete redirection is triggered by the instruction vmfunc, and will not cause a system trap. However, this method needs to insert the instruction vmfunc into the target's code segment. It is suitable for situations where the source code can be obtained.

4.1 Dynamic adjustment for object state

DisAD uses a dynamic method to adjust the state of the target object, which is shown in Fig. 3. The state of an object includes space characteristics and permission types. The object state adjustment includes space expansion, space contraction and permission control. The conditions that trigger the adjustment include memory permission exceptions, out-of-bounds access, breakpoint access, and some specific events.

DisAD uses the method in section 4.1 to adjust the memory permissions and spaces of the target object. The specific events are realized by setting VMCS fields. For example, DisAD can modify the guest rip in VMCS to achieve control flow redirection, modify guest rsp to achieve stack adjustment, and set mov to cr3 to capture process switching.

Breakpoints can be divided into data points and instruction breakpoints. When setting the instruction

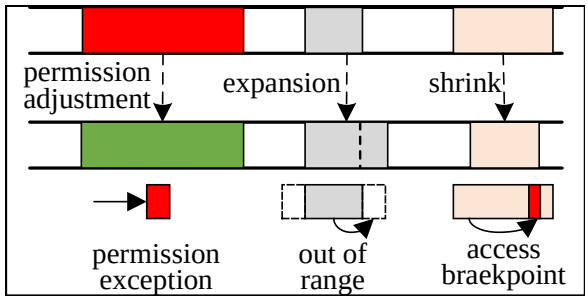

Figure 3 The dynamic adjustment for spaces

breakpoint, we first write the instruction address into the breakpoint address register (Dr0-Dr3). Then the R/W bit corresponding to the breakpoint address register in Dr7 will be set to an execute breakpoint, write breakpoint or read breakpoint. Next, the corresponding LEN bit of Dr7 will be set to 00 (execution length is 1 byte) or 10 (data size is 8 bytes). The access to the breakpoint will cause a system trap, and the access event will be captured by DisAD. DisAD uses the bits B0-B2 of Dr6 to determine the position of the breakpoint. By setting breakpoints, DisAD can achieve byte-granular resource access and instruction execution control to the OS.

5 Runtime Isolation Method

In the scenario where the source code is unavailable, we cannot make changes to the source code. Fortunately, we can still achieve data isolation by manipulating binary code. This section introduces data isolation methods for the execution entities without source code, including heap fence mechanism and stack alternation mechanism.

5.1 Heap fence mechanism

There is no strict isolation between different heaps belonging to a same process, which provides basic conditions for heap overflow. According to the size (128KB), the heap can be divided into large heap and small heap. The address spaces of different large heaps are usually not continuous, while all small heaps are in a continuous address space, which are the targets of heap overflow attacks. In this section, we introduce the heap fence mechanism to achieve isolation and protection for the small heap.

To realize the heap fence mechanism, we first capture the allocation and release of the small heap. The heap allocation and release depend on the library functions malloc (or new) and free respectively. When the application code calls library functions, the control flow needs to pass through PLT and GOT (Global Offset Table). Therefore, DID intercepts the control flow in PLT and modifies it, as shown in Fig. 4. By analyzing the ELF file or setting breakpoints in the dynamic library, DID can obtain the address of the malloc and free entries in the PLT. We define the malloc revision rules as model 3.

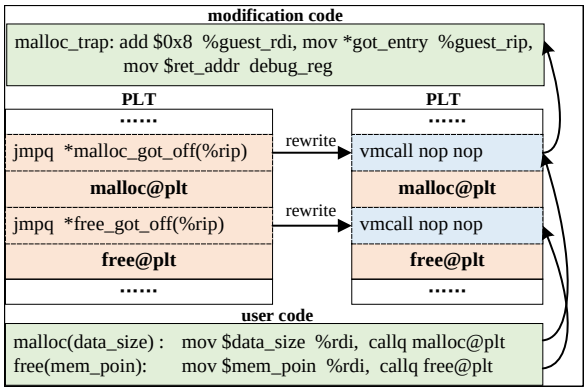

Figure 4 Control flow correction method

1. $f_{revise}(malloc@plt) :: \{vmcall, nop, nop, nop\}$ // revise PLT
2. $\forall instruction \not\prec \{call\ malloc@plt\ in\ guest\}$, // call malloc
3. $ModSwitch[guest \rightarrow host]$,
4. $reg\_revise[rdi = rdi + 8]$, $SetReadBreakpoint(rsp + 8)$

Model 3. Malloc revision rules

DID rewrites the first 6 bytes (instructions of jumping to GOT) of the entries malloc@plt and free@plt in PLT to 0x0f01c19090 (vmcall nop nop nop), which is shown as line 1. Therefore, the process will trigger a system trap due to the execution of the instruction vmcall when allocating or releasing a small heap (lines 2-3). In response to such a system trap, DID modifies the value of the register rdi (the parameter of malloc, that is, the size of the heap requested by the user) in the guest mode to rdi+8 (line 4). The expanded 8 bytes are called fence space and are used to store the fence value. After that, DID writes the address of the GOT entry corresponding to malloc into the rip in the guest mode. Finally, DID sets an execution breakpoint at the return address of malloc.

When the heap allocation is completed, the OS will run in the guest again due to the execution breakpoint. At this time, DID can get the first address of the heap by reading the value stored in rax. DID establishes the management structure of the heap, including the first address, size, and fence value of the heap. When the user releases the heap through free, DID will delete the related management structure.

Based on the above settings, DID establishes a heap fence mechanism, as shown in Fig. 5. We define the heap protection rules as model 4. It should be noted

that this mechanism is only enabled during the dangerous functions running.

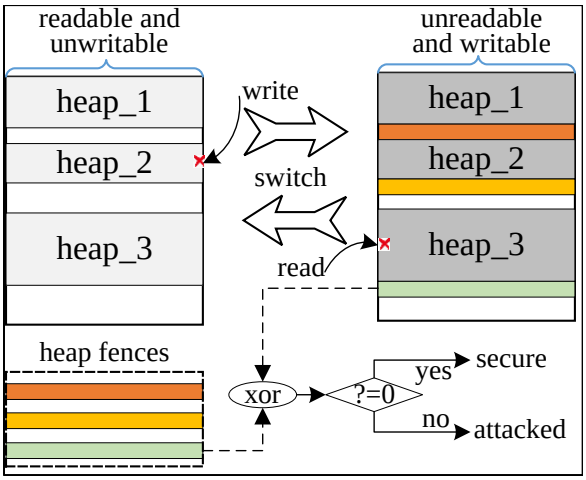

Figure 5 Heap fence mechanism

1. $heap := \{h_1, h_2, \ldots, h_n\}$, $heap\_fence := \{f_1, f_2, \ldots, f_n\}$
2. $fence\_value := \{v_1, v_2, \ldots, v_n\}$ // random fence value
3. $heap\_permission := \{read, write\} := \{(0 \text{ or } 1), (0 \text{ or } 1)\}$
4. $heap\_permission.read \wedge heap\_permission.write == 1$

5. $\exists$ $heap\_permission.write == 0$ and $read\_heap[h_k]$,
6. $ModSwitch[guest \rightarrow host]$,
7. $CheckFence(heap\_fence, fence\_value)$,
8. $heap\_permission.read = 1, heap\_permission.write = 0$,
9. $PullOutFence(heap\_fence)$, $SetReadBreakPoint(f_k)$,
10. $ModSwitch[host \rightarrow guest]$,
11. $\exists$ $heap\_permission.read == 0$ and $read\_heap[h_k]$,
12. $ModSwitch[guest \rightarrow host]$,
13. $heap\_permission.read = 1, heap\_permission.write = 0$,
14. $FillFence(heap\_fence)$, $SetWriteBreakPoint(f_k)$,
15. $ModSwitch[host \rightarrow guest]$.

Model 4. Heap protection rules

DID separates read and write permissions for all heaps in the address space of the execution entity (lines 3-4). When a user reads heap (h_k) data (line 5), DID checks the correctness of all the fence values (lines 6-7). Any inconsistency will lead to a general exception protection. If all the value keeps unchanged, DID closes all heap write permissions and only retains its read permissions (line 8). After that, DID will pull out all the fence value and add a reading breakpoint at the fence address of the current heap to prevent out-of-bounds reading (line 9).

When the user needs to write data to the heap, the OS will trigger an EPT violation and fall into the host (line 11-12). Then, DID closes the read permission of all heaps in the address space of the execution entity, and opens the write permission (line 13). Then it fills the heap fence (f_k) with a random value (v_k), and sets a writing breakpoint at f_k. Therefore, heap overflow and out-of-bounds access occurring in the current heap will be detected and prevented by DID.

The heap management structure is completely managed by DID. They are stored in the host space and only allow DID to access them. Since the read permission and write permission cannot be enabled at the same time, and all the heap fences have been pulled out when an entity read them, any user in guest cannot read the fence value. This makes it impossible for an attacker to bypass the heap fence mechanism by probing or rewriting the fence value.

5.2 Stack alternation mechanism

Stack overflow are often used for control flow hijacking. CRAs can redirect the control flow to the gadget chain by rewriting the control data on the stack. Since gadgets are extremely demanding on the code forms, the attacker needs a large amount of binary code as the source of gadget construction. Therefore, the large-scale dynamic library has become a breeding ground for attackers to build gadgets.

Attackers exploit the stack overflow to tamper with the control data. Then the control flow can be redirected to the gadget in the library when it returns or jumps. To solve this problem, DID establishes a stack alternation mechanism, as shown in Fig. 6. We define the operation rules of this mechanism as model 5. This mechanism is also only enabled during the dangerous functions running.

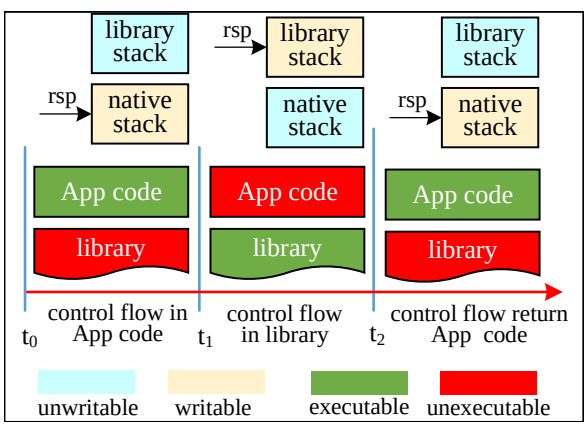

Figure 6 Stack alternation mechanism

1. $stack := \{lib\_stack, nat\_stack\}$, $code := \{lib\_code, app\_code\}$
2. $Permission := \{executable, unexecutable\} = \{1, 0\}$
3. $PermitOf(lib\_code) \wedge PermitOf(app\_code) == 0$

4. $\forall instruction \not\in lib\_code, make\ rsp\ points\ to\ lib\_stack$
5. $PermitOf(lib\_code) = 1, PermitOf(app\_code) = 0$
6. $\forall instruction \not\in app\_code, make\ rsp\ points\ to\ nat\_stack$
7. $PermitOf(lib\_code) = 0, PermitOf(app\_code) = 1$

Model 5. Stack alternation rules

When the control flow is in the application code, the status of the execution entity is shown as t0-t1 in Fig. 6. When the control flow jumps to the library (lib_code) from the application code (app_code), DID opens the execute permission of the library and closes the execute permission of the application code (lines 4-5). After that, DID enables the new stack (lib_stack) and closes the read and write permissions of the original stack (nat_stack). When the control flow returns, DID restores the status of each object to the initial value (lines 6-7).

When library functions need to use local variables as pointer parameters, lib_stack will have a dependency on nat_stack. DID can determine whether the dependency exists according to whether the value of the parameter register (rdi, rsi, etc.) points to nat_stack. When the library function accesses the nat_stack data pointed to by the parameter register, DID will enable the read and write permissions of nat_stack and disable the read and write permissions of lib_stack. When the library function accesses the lib_stack again, the permissions of the two stacks are switched with each other. This method can ensure the correctness of data access, and can also prevent attackers from tampering with and probing stack data.

When an attacker directly calls a library function with a stack overflow vulnerability in the application code, the lib_stack will be enabled. At this time, the attacker can neither read the data in the nat_stack nor tamper with it.

Moreover, the stack alternation mechanism also helps to break the connection the gadget chain. If an attacker tampered with the control data on the stack through application code instead of a library function, the control flow will smoothly jump to the gadget in the library according to the attacker's intention. In order to connect all the gadgets together, the attacker needs to fill the addresses of all gadgets into the nat_stack before the gadget jumps. Then the dispatcher-gadget[37] can connect all the gadgets through the addresses stored on nat_stack. However, due to the different permissions and spaces between nat_stack and lib_stack, after the control flow jumps to the library, the attacker will not be able to read any information stored on nat_stack. This will cause the break of the gadget chain, causing the attack to fail.

6 Hybrid Isolation Method

In this section, we introduce a more fine-grained data isolation mechanism for the objects with source code.

6.1 Heap owner mechanism

The data in the heap not only faces the challenge of overflow attacks, but also has the risk of being stolen. In particular, private information such as secret keys stored in the heap can be obtained by attackers through techniques such as memory leakage and process cloning. To enhance the privacy of a specific heap, we propose a heap owner mechanism, as shown in Fig. 7. We define the operation rules of heap owner mechanism as model 6.

To realize the heap owner mechanism, DID needs to take over the target heap allocation in source code. It allocates the private heap to store secret data into one or several specific memory pages. After that, a new EPT page table will be created to index the private heap. At the same time, DID records the process ID (PID) and the address of the function to access the private heap. Within the recorded function, one or several statements that access the private heap are called a code block. DID establishes the owner relationship of the target heap from the three granularities: process (p_j), function (f_j) and code block (c_j), which is shown as line 4. Each code block has and only has one function corresponding to it, not vice versa (line 5). In the heap owner mechanism, in addition to the main EPT page table (ept0), each set of EPT page tables corresponds to a private heap, and each code block (c_j) has a set of owner information (owner_h_j) corresponding to it (line 6).

1. $heap := \{h_1, h_2, \ldots, h_n\}$, $code\_block := \{c_1, c_2, \ldots, c_k\}$
2. $func\_addr := \{f_1, f_2, \ldots, f_i\}$, $pid := \{p_1, p_2, \ldots, p_m\}$
3. $ept\_list := \{ept_0, ept_1, ept_2, \ldots, ept_n\}$
4. $current\_state := \{p_x, f_x, c_x\}$, $owner\_h_j := \{p_j, f_j, c_j\}$
5. $code\_block \mapsto func\_addr \mapsto pid$
6. $c_j \bowtie owner\_h_j, h_j \bowtie ept_j$

---

7. $\forall instruction \triangleright c_j, enable(ept_j)$
8. $\exists data\ access \not\prec h_j$, has $ModSwitch[guest \rightarrow host]$
9. $CheckOwner(owner\_h_j, current\_state)$

Model 6. Heap owner rules

When a process attempts to access the private heap, DID enables the instruction vmfunc in source code to switch EPT before its code block is executed. Then, the private heap appears in the address space of the current process (line 7). If the data access to private heap (h_j) triggers a system trap, DID detects whether the current owner information (including the current process number and function address, called owner_state) is consistent with the owner information (owner_h_j) of the private heap being accessed. Any non-owner control flow (with different processes, different functions, or different code blocks) cannot access the private heap.

After the code block is executed, vmfunc is called again to switch back to the original EPT, in which the private heap is invisible. Based on this method, DID establishes a differentiated permissions and spaces for each private heap, so that the private heap only allows to be accessed by the specified code block.

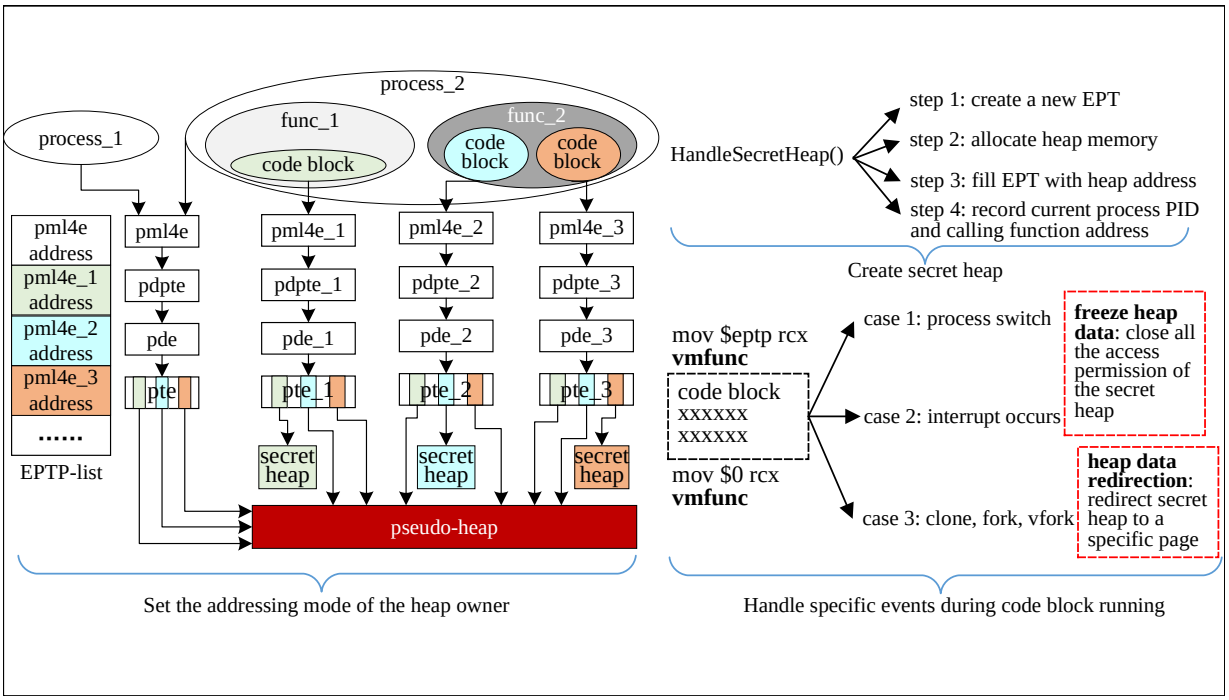

Figure 7 Heap owner mechanism

In different EPT page tables, except that the corresponding private heap can be correctly addressed, the addressing items of other private heaps are all redirected to a pseudo heap. At the same time, the two memory pages adjacent to the first and last addresses of the private heap will be redirected to a physical page that is unreadable and unwritable through EPT redirection. Therefore, any probing and out-of-bounds access to the private heap will trigger a system trap and be captured by DID.

To further protect the private heap, DID monitors process switching, interruption, process cloning and breakpoint access during the running of the process containing private heaps. If a process switching or interruption event occurs during the code block running, DID will immediately freeze the private heap (close its read and write permissions). If a process cloning event occurs during the execution of the code block, DID redirects the private heap to the pseudo heap. Therefore, all execution entities other than the heap owner (the code block corresponding to private heap) cannot obtain the private heap information. When the control flow returns to the code block, the OS will fall to host from guest. Then DID checks whether the control flow has the permission to access the private heap according to the PID, function address, and code block address. If any item does not match, the access request will be rejected.

The attacker destroys the integrity of the function stack frame through a dangerous function. Therefore, we need to ensure the integrity of the stack data for the function when it calls a dangerous function. DID proposes a stack jump mechanism for the objects with source code, as shown in Fig. 8. We define the operation rules of stack jump mechanism as model 7.

6.2 Stack jump mechanism

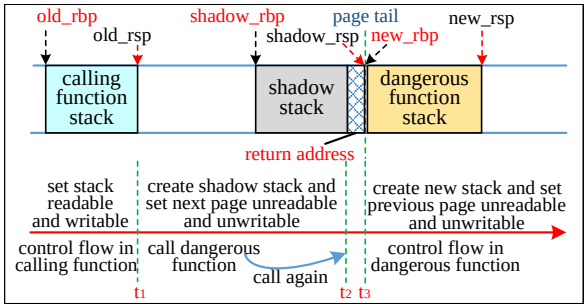

Figure 8 Stack jump mechanism

1. $stack := \{C_{stack}, S_{stack}, D_{stack}\}$ // three stacks
2. $function := \{C_{function}, D_{function}\}$ // two functions
3. $stack_{page} := \{pre_{page}, next_{page}\}$ // two types of pages
4. $C_{stack} \propto pre_{page}, S_{stack} \propto pre_{page}, D_{stack} \propto next_{page}$

5. $I[C_{function}] \triangleright I[D_{function}], Close(next_{page}), enable(D_{stack})$

6. $dataaccess \not\prec next_{page}$ // data access in the new stack
7. $Open(next_{page}), Close(pre_{page}), enable(D_{stack})$
8. $I[D_{function}] \triangleright I[C_{function}], Open(pre_{page}), Enable(C_{stack})$

Model 7. Stack Jumping Rules

There are three types of stack frames including the stack of the calling function (C_{stack}), the shadow stack (S_{stack}), and the stack of the dangerous function (D_{stack}) in the stack jump mechanism, as shown in line 1. DID divides all the functions into dangerous functions and calling functions(line 2). Taking the page to which the C_{stack} belongs as the dividing line, the high-address memory is called pre_{page} (including the current memory page), and the low-address memory is called $next_{page}$ (line 3). C_{stack} and S_{stack} belong to pre_{page}, and D_{stack} belongs to $next_{page}$ (line 4).

When the control flow jumps from the calling function ($C_{function}$) to the dangerous function ($D_{function}$), DID enables S_{stack} and closes the read and write permissions of $next_{page}$(line 5). After that, when the execution entity accesses D_{stack}, DID opens the read and write permissions of $next_{page}$ and enables D_{stack} (lines 6-7). At the same time, DID also closes the read and write permissions of pre_{page}. When the control flow returns, the permissions of C_{stack} will be restored to the state before the dangerous function is called(line 8).

To implement the stack jump mechanism, it is necessary to find dangerous functions derived from application code and libraries. The dangerous functions that have been discovered are indicated in various vulnerability libraries provided by National Vulnerability Database[1] and China National Vulnerability Database of Information Security[2]. Therefore, DID can identify the dangerous function contained in the source code based on the vulnerability library information. For application code spaces, DID can identify dangerous functions through static analysis, and can even treat all functions mentioned in Table 1 as dangerous functions. DID adds an embedded instruction vmcall before the statement call dangerous function to capture the attempt of the dangerous function call. Before the dangerous function is executed, the OS will fall into the host.

Then, DID creates a shadow stack frame and make the registers *rsp* and *rbp* point to the shadow stack frame. The *rsp* in the shadow stack frame (*shadow_rsp*) points to the start address of the current page. If the shadow stack frame covers other stack frames, the shadow stack frame extends downward by 4KB. To ensure the shadow stack frame is the same size as the calling function stack frame, it is necessary to make $shadow\_rbp - shadow\_rsp = old\_rbp - old\_rsp$. After that, DID sets the next memory page as unreadable and unwritable.

When the calling function calls the dangerous function, its return address will be filled into the first 8 bytes of the current page (determined by the top-down growth characteristics of the stack). Then, the dangerous function will take the next page as a stack frame, which is unreadable and unwritable. As a result, a system trap will be triggered when a stack access occurs. Next, the current page is set to be readable and writable, and the previous page is set to be unreadable and unwritable. At this point, the new stack frame is enabled for the dangerous function.

If the dangerous function depends on the stack frame data of the calling function, it will access these data during execution. When the dangerous function is being called, it uses the *new_rbp* or *new_rsp* to locate the stack data on the stack frame of the calling function. At this point, the data location calculated based on the relative offsets is located on the shadow stack, which is unreadable and unwritable. The dangerous function's access to the data on shadow stack frame will trigger a system trap. If the exception address is not the first 8 bytes of the previous page, it is considered that the dangerous function depends on the shadow stack frame. If the first 8 bytes of the previous page is being accessed and the current instruction is not ret, the current operation will be considered malicious and be blocked. When dependency occurs, the shadow stack frame does not contain any stack frame data of the calling function. Therefore, DID needs to copy the data of the original stack frame of the calling function to the shadow stack frame. After that, DID opens the read and write permissions of the shadow stack frame and closes the read and write permissions of the dangerous function stack frame. After the access is completed (the dangerous function stack frame generates an EPT exception), the DID exchanges the permissions of the shadow stack frame and the dangerous function stack frame.

After the dangerous function is executed, the return address stored in the previous page will be read through the instruction ret. DID can identify the event through the abnormal address and the abnormal instruction. At the same time, DID can use the dirty bit of EPT to determine whether the dangerous function has written data to the shadow stack frame. It will write back the changed shadow stack to the stack frame of the calling function to ensure the data consistency. Finally, DID restores *rsp* and *rbp* to their initial values.

Through stack jump mechanism, DID realizes the stack data isolation with function granularity. Any stack overflow generated by dangerous functions can

[1]https://nvd.nist.gov/
[2]https://www.cnnvd.org.cn/

be detected. That is, the attacker cannot use the dangerous functions to destroy the stack data of the calling functions.

7 Discussion on DID

For the scenarios of no source code, DID isolates stack data and heap data by analyzing and operating the binary code. Since no source code is required, the generality of heap fence mechanism and stack alternation mechanism is good. For the scenarios with source code, DID uses a combination of source code operation and runtime tracking to achieve better isolation. However, this method requires that we must be able to implement corrections to the source code. Otherwise, the method will fail to work.

In fact, it is still possible to apply the heap owner mechanism and stack jump mechanism in the case of no source code. For the heap owner mechanism, we need to determine the target heap and its owner information. For the stack jump mechanism, we need to recognize and mark the dangerous functions. This information can be achieved using instruction tracking and binary analysis.

After that, we can migrate the target heap to a private memory page, and use binary rewriting to redirect the heap owner's data access to the private page. For dangerous functions, we mark it with trapping instructions such as INT 3 or vmcall. After capturing the trap instructions, we enable the stack jump activity, similar to user space.

However, none of the current binary tracking methods can consider efficiency and effect together. For example, we tracked, analyzed, and redirected all heap access instructions of *OpenSSH*, which caused the heap access speed to slow down by more than 10 times, and the remote login latency increased by 173%. Therefore, we still need to find a faster and more effective binary analysis method.

In this paper, we mainly discuss the isolation method in user space. For the kernel space, the four mechanisms proposed by DID are not applicable.

8 Security And Performance Evaluation

We conduct all the experiments on a Dell T440 server, which is equipped with two 10-core Intel Xeon silver 4210 2.2GHz CPUs and 128GB memory. The OS is Ubuntu-16.04 with kernel 4.4.1.

8.1 Security evaluation

In this section, we evaluate the effect of DID on the heap and stack. We use the application's out-of-bounds access to simulate the attacker's data theft and tampering. Then we observe the defensive effect of DID.

8.1.1 Heap data protection

The threats of heap data are divided into internal threat and external threat. The internal threat refers to data leakage or destruction caused by the execution entity's internal code, such as heap overflow vulnerabilities. External threats refer to the malicious impact of other execution entities on the target process, such as secret key scanning tools. In this section, we use *OpenSSH + OpenSSL* as the test objects, and the schematic diagram of defense effect is shown in Fig.9.

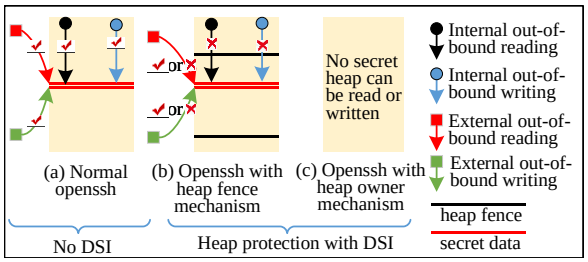

Figure 9 Heap protection test

To test the defense effect of DID against internal threats, we rewrite the heap allocation function OPENSSL_malloc in *OpenSSL* and add a heap out-of-bounds access function in it. The added function can continuously read data in multiple heaps from low address to high address. The result shows that the heap allocated by OPENSSL_malloc can be protected by both heap owner mechanism and heap fence mechanism.

To test the effect of DID's defense against external threats, we create an LKM, which can read and rewrite the secret key data on the heap like a secret key scanning tool. The results show that the LKM can access heap data when there is no DID. Under the protection of the heap fence mechanism, the LKM can be prevented by heap owner mechanism when it accesses the secret data. In contrast, only when the LKM accesses multiple heaps at once, heap fence mechanism can find the illegal operation.

8.1.2 Stack data protection

Unlike the heap, the stack is only used to store local variables (such as local arrays) and control data (such as return addresses) of some functions, rather than private data. Malware can use overflow vulnerabilities of local variables to tamper with control data, thereby hijacking control flow.

To test DID's stack protection mechanism, we design application functions and library functions with stack overflow risks. These functions can implement stack overflow attacks by copying data between arrays (local variables). The experiment results are shown

in Fig. 10. The results show that the stack alternation mechanism can achieve strong isolation for stack frames between application functions and library functions. However, it does not have any defense effect on stack overflow attacks between functions with the same attribute (both the calling function and dangerous function belong to library code). In contrast, the stack jump mechanism can prevent stack overflow attacks when a specified dangerous function is called.

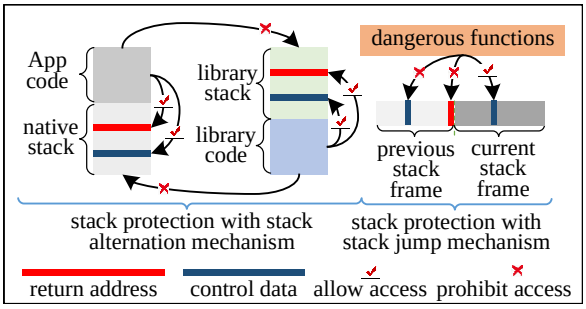

Figure 10 Stack protection test

8.1.3 Safety analysis

Analysis of Heap Data Protection. The heap fence mechanism uses fence value to detect out-of-bounds access of internal threats, and prevents the illegal operations by injecting a general protection exception. It dynamically adjusts the read and write permissions of the heap according to the access requirements (read or write) of the process to which the heap belongs. When an attacker and the process to which the heap belongs have the same heap access requirements simultaneously, it can successfully steal or tamper with the secret key. Otherwise, the access will cause an EPT exception.

The heap owner mechanism restricts the execution entity that can access the heap must be a heap owner. No control flow other than the owner can obtain the address of the heap. Therefore, neither internal threats nor external threats can read and write the private heap data.

Stack Data Protection Analysis. The stack alternation mechanism reconstructs the permissions and spaces of the stack according to the code attribute (application code or library code). According to the control flow locations, DID switches different stack spaces for the process. Therefore, there is strict isolation between function stack frames with different locations. However, the code with the same attributes still use the same stack, resulting in no isolation effect between their stack frames.

In contrast, the stack jump mechanism has a finer protection granularity. This mechanism establishes a new function stack frame for the specified dangerous function, and performs differentiated management for different stack frames. This makes dangerous functions unable to tamper with control data outside their stack frames.

Heartbleed Defense Analysis. The essence of the Heartbleed attack is no check for the dangerous function parameters, which leads to an out-of-bounds access to the heap data. Under the protection of the heap fence mechanism, Heartbleed's out-of-bounds access will trigger a read breakpoint at the fence value position, which will be captured and blocked by DID. For the heap data protected by the heap owner mechanism, all out-of-bounds access of Heartbleed will end in failure. Because the control flow does not belong to the heap owner.

CRA Defense Analysis. CRA uses stack overflow vulnerabilities to tamper with control data (including return address and jump address) to intercept control flow. If an attacker uses application code to call library functions with stack overflow vulnerabilities to tamper with control data, which is the most common attack method, the stack alternation mechanism can completely prevent such an attack. However, if the stack overflow is done in application code or library completely, the stack alternation mechanism has no protection effect. The stack jump mechanism can prevent the out-of-bound access to different stack frames of dangerous functions. However, if the stack overflow occurs in one stack frame (such as tampering with the function pointers stored in its own stack frame), the stack overflow cannot be detected.

Fortunately, most of the current attacks are not done within a single function stack frame. It is determined by the strict gadget forms needed by CRAs. To construct multiple gadgets with a specific form, attackers often use application functions to connect multiple library code snippets together directly or indirectly.

Compared with existing methods, DID has advantages in certain aspects. For the existing software-based methods[32, 33, 35, 34], the target objects containing sensitive data must be known before they are deployed. That is to say, they must know which data are vulnerable. However, for the closed-source targets, their sensitive data is always unknown, which leads to work failure of the software-based methods. In contrast, DID can protect the targets containing source code, and it can also protect the closed source targets. For existing hardware-based methods[27, 30, 31], they require the position and size of the target object to be relatively fixed. If the size and position of protected objects are dynamically changing, their protection effect will be greatly affected. Unlike existing hardware-based methods, DID can dynamically track the allocation and recycling of target objects, thus maintaining

good protection against dynamically changing heaps and stacks.

8.2 Performance evaluation

In this section we evaluate the impact of DID on the performance of OS and applications. All the results were normalized with the native OS as the standard.

8.2.1 Impact of DID on OS

Lmbench is used to test DID's impact on OS performance, and the experiment results are shown in Fig. 11. The results show that DID slows CPU processing speed by 1.9% and slows context switching speed by 5.8% on average. It also increases communication latency by 3.2%, file operation latency by 5.6%, and memory latency by 1.6%.

To further test the impact of DID on OS, we introduce some microbenchmarks, as shown in Table 3. We found that a system trap (VMEntry/VMExit) takes much longer than other operations. In comparison, the EPT switching generated by the instruction vmfunc takes less time. This is because vmfunc does not cause the system trap. Therefore, we conclude that the system trap caused by DID is the main factor affecting OS performance.

Table 3 Micro benchmarks.

call user function	call library function	system call	vmfunc	VMEntry /VMExit
3.21 ns	4.17 ns	59.79 ns	112.37 ns	584.39 ns

The type of system traps can be divided into two types: unconditional traps and conditional traps. In the guest mode, the instructions cpuid, gettsec, invd, xsetbv, and all VMX instructions except vmfunc will cause system traps unconditionally. Conditional traps are triggered by specific events set by DID, including breakpoints access, general protection exceptions, EPT exceptions, and dangerous function returns. After DID processes the trap events, the OS will switch back to the guest mode again. In the whole process, mode switching and event handling will affect the OS performance. During the Lmbench test, we found that the cpuid instruction caused by the context switching is executed very frequently. Therefore, the high-frequency system traps caused by this instruction generates a larger performance overhead.

In practice, the heap fence mechanism, stack alternation mechanism, and stack jump mechanism are not always enabled. They are enabled if and only if a dangerous function containing variable arguments is running. Therefore, they do not introduce excessive overhead. For the heap owner mechanism, it only acts on the heap containing private data. If only the boundaries of the heap need to be protected, the heap fence mechanism is enough.

8.2.2 The impact of DID on applications

This section uses ssh remote login delay to measure the delay overhead generated by DID. We set the heap created by OPENSSL_malloc in *OpenSSL* as the owner heap for the heap owner mechanism, and use the functions do_log and login_write in *OpenSSH* as dangerous functions for the stack jump mechanism.

The heap delay test shows that the heap fence mechanism increases the ssh remote login delay by 7.2ms, and the delay overhead is 6.3%. The heap owner mechanism increases the login delay by 1.7ms, and the delay overhead is 1.5%. The stack delay test shows that the stack alternation mechanism increases the login delay by 13.5ms, and the delay overhead is 11.8%. The stack jump mechanism increases the login delay by 8.7ms, and the delay overhead is 7.6%.

To test the impact of DID on application execution speed, this section customizes a heap access application HeapApp and stack access application StackApp to simulate the data access. HeapApp contains two 4KB heaps, and its control flow will alternately read and write data in the two heaps. For the heap fence mechanism, we test the impact of the read and write switching numbers on the running speed. Then, we use the heap owner mechanism to protect one of the two heaps in HeapApp. The number of alternating accesses to the two heaps is the number of owners switching. StackApp can call library functions and application functions cyclically. For the stack alternate mechanism, we control the number of stacks switching times by adjusting the number of cycles of library function calls. For the stack jump mechanism, we regard an application function as a dangerous function and control the number of stacks jumping by adjusting the number of loop calls.

The performance test result of the heap protection mechanism is shown in Fig. 12. When the number of switching is 1000, the heap fence mechanism and the heap owner mechanism slow down the process execution speed by 1.7% and 5.7%, respectively. When switching 1 million times, the two values increased to 4.3% and 11.2%, respectively. After that, the process execution speed will gradually slow down as the number of switching increases.

The performance test result of the stack protection mechanism is shown in Fig. 13. The results show that the impact of stack alternation mechanism and stack jump mechanism on the execution speed of the process is less than 1% when the calling number of library functions or dangerous functions is less than 10,000. When the number of calls is greater than 100,000, the process execution speed will be severely affected.

When a dangerous function is being executed, the heap fence mechanism will be enabled, and every

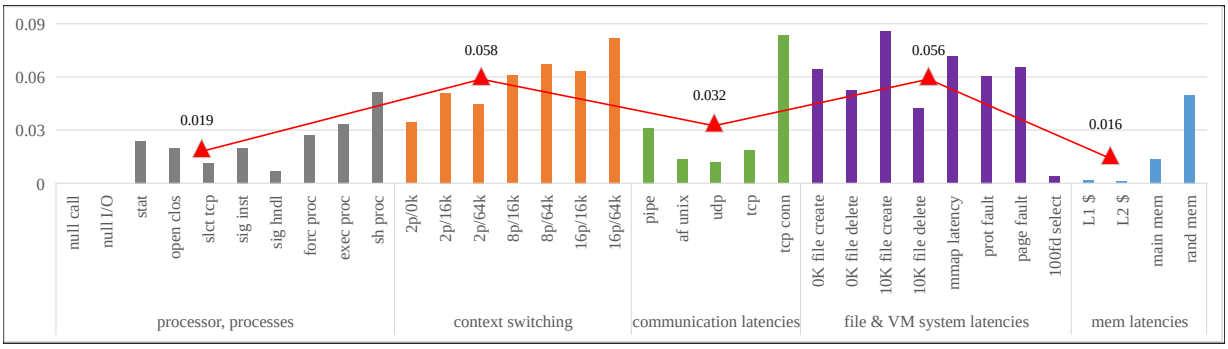

Figure 11 Lmbench test results. The ordinate represents the performance degradation factor, and the abscissa represents the test items.

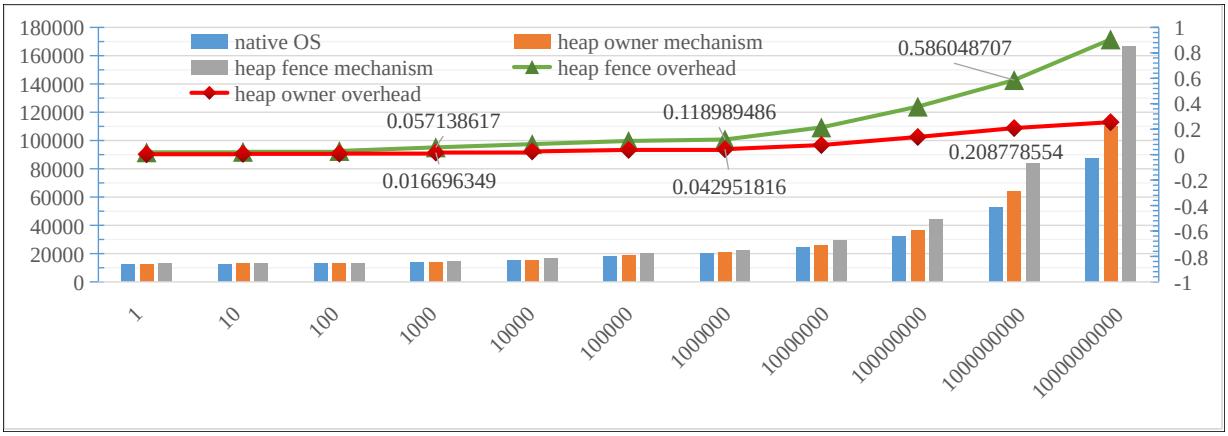

Figure 12 The impact of heap protection mechanism on application execution speed. The ordinate on the left represents the running time (ms) of HeapApp, which corresponds to the bar graph. The ordinate on the right represents the speed degradation factor, which corresponds to the line graph. For the heap fence mechanism, the abscissa represents the number of switching between heap read and write during the application running; For the heap owner mechanism, the abscissa represents the number of the heap owner switching during the application running.

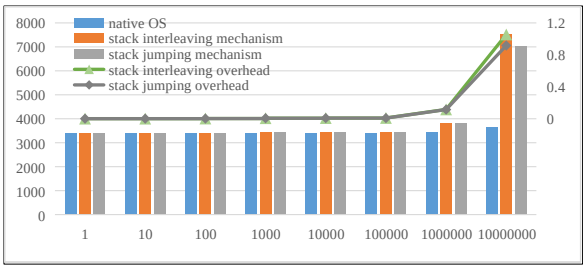

Figure 13 The impact of stack protection mechanism on application running speed. The ordinate on the left represents the running time (ms) of StackApp, which corresponds to the bar graph. The ordinate on the right represents the speed degradation factor, which corresponds to the line graph. For the stack alternate mechanism, the abscissa represents the number of library function calls; For the stack jumping mechanism, the abscissa represents the number of risk function calls.

switching between read and write operations will cause one system trap. For the heap owner mechanism, the creation of a private heap will cause a system trap. After that, every switch of the heap owner will generate two EPT switches.

For the stack alternation mechanism, application code will trigger a system trap when it calls a dangerous function. Afterwards, the control flow returning from the library to the application code will trigger a system trap again. Therefore, a complete switching between the application code and the library will cause two system traps. For the stack jump mechanism, the execution of a dangerous function will cause a system trap. When the dangerous function returns to its calling function, the OS will fall into host again. Therefore, a complete execution of the dangerous function causes two system traps.

The system traps will interrupt the current behavior of the execution entity, and the execution will continue after the DID has processed the trap events. Therefore, the delay and execution speed of the execution entity will be affected. The EPT switching speed is significantly faster than the system trap, so the overhead it introduces is also smaller.

9 Conclusion

This paper proposes a method DID to isolate heap data and stack data. DID constructs a dynamic adjustment framework for permissions and spaces, which is used to adjust the permissions and spaces of the target object. Based on this framework, DID proposes a heap fence mechanism, a heap owner mechanism, a stack alternation mechanism, and a stack jump mechanism. Experiments and analysis show that these mechanisms have good defense effects against heap overflows and stack overflows. It should be noted that DID is fully portable to a virtualization enabled environment, as long as the processor supports VMX and EPT.

However, DID still has some limitations. First, DID only supports the Linux, which is invalid for closed-source systems such as Windows. Second, DID only has an isolation effect on the heap and stack in the user space, and is invalid on the kernel space.

Declarations

Ethical Approval and Consent to participate

Not applicable.

Consent for publication

All authors, YongGang Li, Zhi Qu, and Yeh-Ching Chun, have read and approved the final manuscript and consent to its publication.

Availability of supporting data

The compiled and refined data and results of this manuscript should be avail able upon request by the authors.

Authors' contributions

YongGang Li conceived the idea for the research and provided guidance throughout the study. Zhi Qu conducted extensive literature review and identified the limitations of existing isolation methods for heap and stack data storage in operating systems (OS). Yeh-Ching Chung contributed to the design and implementation of the proposed isolation method, termed DID (Dynamic Isolation of Data). All authors collectively reviewed and revised the manuscript, ensuring its intellectual integrity, accuracy, and clarity. YongGang Li provided oversight throughout the writing process, while Zhi Qu and Yeh-Ching Chung contributed significantly to the refinement of the paper's technical content.

Competing interests

The authors declare no competing interests.

Funding

This work is supported by Jiangsu Province Double-innovation Doctor Project and Huawei Corporation under contract TC20220913039.

Author details

[1]School of Computer Science and Technology, China University of Mining and Technology, XuZhou. [2]School of Data Science, Chinese University of Hong Kong, ShenZhen, ShenZhen.

References

1. Chen, W., Poor, H.V.: Caching with time domain buffer sharing. IEEE Trans. **67**(4), 2730–2745 (2019). doi:10.1109/TCOMM.2018.2884973
2. Wang, K., Li, J., Fu, Z., Chen, T.: Irepf: An instruction reorganization virtual platform for kernel stack overflow detection. Secur. Commun. Netw. **2022** (2022). doi:10.1155/2022/7645210
3. Liu, Y., Wang, Y., Feng, H.: Poaguard: A defense mechanism against preemptive table overflow attack in software-defined networks. IEEE Access **11**, 123659–123676 (2023). doi:10.1109/ACCESS.2023.3330224
4. Patwardhan, A., Jayarama, D., Limaye, N., Vidhale, S., Parekh, Z., Harfoush, K.: Sdn security: Information disclosure and flow table overflow attacks. IEEE Glob. Commun. Conf. , Waikoloa, HI, DEC 09-13 (2019)
5. Harvey-Lees-Green, N., Biglari-Abhari, M., Malik, A., Salcic, Z.: A dynamic memory management unit for real time systems. 20th IEEE International Symposium on Real-Time Distributed Computing(ISORC), Toronto, CANADA, MAY 16-18 **19**, 84–91 (2017)
6. Lee, S., Kang, H., Jang, J., Kang, B.B.: Savior: Thwarting stack-based memory safety violations by randomizing stack layout. IEEE Trans., 2559–2575 (2022)
7. Duta, V., Giuffrida, C., Bos, H., van der Kouwe, E.: Pibe: Practical kernel control-flow hardening with profile-guided indirect branch elimination. ASPLOS XXVI: TWENTY-SIXTH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 743–757 (2021)
8. Shrivastava, R.K., Concessao, K.J., Hota, C.: Code tamper-proofing using dynamic canaries. 25th APCC, 238–243 (2019)
9. Zhang, T., Cai, M., Zhang, D., Huang, H.: Sebrop: blind rop attacks without returns. Front. Comput. Sci. **16** (2022). doi:10.1007/s11704-021-0342-8
10. Wang, C., Chen, B., Liu, Y., Wu, H.: Layered object-oriented programming: Advanced vtable reuse attacks on binary-level defense. IEEE Trans. Inf. Forensics Secur. **14**, 693–708 (2019). doi:10.1109/TIFS.2018.2855648
11. An, Z., Wang, W., Li, W., Li, S., Zhang, D.: Securing embedded system from code reuse attacks: A lightweight scheme with hardware assistance. Micromachines **14** (2023). doi:10.3390/mi14081525
12. Saileshwar, G., Boivie, R., Chen, T., Segal, B., Buyuktosunoglu, A.: Heapcheck: Low-cost hardware support for memory safety. ACM Trans. Archit. Code Optim. **19** (2022). doi:10.1145/3495152
13. Boivie, R., Saileshwar, G., Chen, T., Segal, B., Buyuktosunoglu, A.: Hardware support for low-cost memory safety. DSN, 57–60 (2021)
14. Lin, Y., Tang, X., Gao, D.: Safestack+: Enhanced dual stack to combat data-flow hijacking. ACISP, 95–112 (2017)
15. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. USENIX Secur. Symp., 63–78 (1998)
16. Huang, H., Lu, Y., Zhu, K., Zhao, J.: Canaryexp: A canary-sensitive automatic exploitability evaluation solution for vulnerabilities in binary programs. Applied Sciences-Basel **13** (2023). doi:10.3390/app132312556
17. Tsai, T., Singh, N.: Libsafe: Transparent system-wide protection against buffer overflow attacks. DSN, 541 (2002).

doi:10.1109/DSN.2002.1028963. IEEE Comp Soc Tech Comm Fault Tolerant Comp; IFIP Working Grp 10 4; DARPA; LAAS CNRS; Univ Michigan
18. Shen, Z., Chen, W.: A survey of research on runtime rerandomization under memory disclosure. IEEE Access **7**, 105432–105440 (2019). doi:10.1109/ACCESS.2019.2931707
19. Jin, H., Liu, B., Du, Y., Zou, D.: Boundshield: Comprehensive mitigation for memory disclosure attacks via secret region isolation. IEEE Access **6**, 36341–36353 (2018). doi:10.1109/ACCESS.2018.2835838
20. Fu, Y., Lin, J., Feng, D., Wang, W., Wang, M., Wang, W.: Regkey: A register-based implementation of ecc signature algorithms against one-shot memory disclosure. ACM Trans. Embed. Comput. **22** (2023). doi:10.1145/3604805
21. Shi, J., Guan, L., Li, W., Zhang, D., Chen, P., Zhang, N.: Harm: Hardware-assisted continuous re-randomization for microcontrollers. IEEE 7th EuroS&P, 520–536 (2022). doi:10.1109/EuroSP53844.2022.00039. IEEE; IEEE Comp Soc
22. Luo, L., Shao, X., Ling, Z., Yan, H., Wei, Y., Fu, X.: faslr: Function-based aslr via trustzone-m and mpu for resource-constrained iot systems. IEEE Internet Things J. **9**, 17120–17135 (2022). doi:10.1109/JIOT.2022.3190374
23. Mow, W.-L., Huang, S.-K., Hsiao, H.-C.: Laeg: Leak-based aeg using dynamic binary analysis to defeat aslr. IEEE DSC (2022). doi:10.1109/DSC54232.2022.9888796. IEEE; IEEE Reliabil Soc; Univ Exeter; Scottish Informat & Comp Sci Alliance; Univ Kent; Inst Cyber Secur Soc; Chartered Inst IT, Edinburgh Branch
24. Lu, K., Nuernberger, S., Backes, M., Lee, W.: How to make aslr win the clone wars: Runtime re-randomization. NDSS (2016). doi:10.14722/ndss.2016.23173
25. Li, Y., Cai, J., Bao, Y., Chung, Y.-C.: What you can read is what you can't execute. Comput Secur. **132** (2023). doi:10.1016/j.cose.2023.103377
26. Curtsinger, C., Berger, E.D.: Stabilizer: Statistically sound performance evaluation. ACM Sigplan Not. **48**, 219–228 (2013). doi:10.1145/2499368.2451141
27. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D.: Erim: Secure, efficient in-process isolation with protection keys. Usenix Security Symposium, 1221–1238 (2019). USENIX Assoc; Facebook; Microsoft; Intel; Avast; Baidu; Cisco; Google; King Abdullah Univ Sci & Technol; Paloalto Networks; Dropbox; IBM Res; Kaspersky; USC Viterbi, Sch Engn, Informat Sci Inst; Bloomberg; NetApp
28. Park, S., Lee, S., Xu, W., Moon, H., Kim, T.: libmpk: Software abstraction for intel memory protection keys. USENIX, 241 (2019)
29. Liu, Y., Zhou, T., Chen, K., Chen, H., Xia, Y.: Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. ACM CCS, 1607–1619 (2015). doi:10.1145/2810103.2813690. ACM SIGSAC; Assoc Comp Machinery
30. Proskurin, S., Momeu, M., Ghavamnia, S., Kemerlis, V.P., Polychronakis, M.: xmp: Selective memory protection for kernel and user space. IEEE SP, 563–577 (2020). doi:10.1109/SP40000.2020.00041. IEEE; IEEE Comp Soc
31. Sartakov, V.A., O'Keeffe, D., Eyers, D., Vilanova, L., Pietzuch, P.: Spons & shields: Practical isolation for trusted execution. ACM VEE, 186–200 (2021). doi:10.1145/3453933.3454024. Assoc Comp Machinery; ACM SIGPLAN; ACM SIGOPS; Google; USENIX
32. Carr, S.A., Payer, M.: Datashield: Configurable data confidentiality and integrity. ACM ASIA CCS, 193–204 (2017). doi:10.1145/3052973.3052983. Assoc Comp Machinery; ACM SIGSAC; New York Univ Abu Dhabi
33. Brahmakshatriya, A., Kedia, P., McKee, D.P., Garg, D., Lal, A., Rastogi, A., Nemati, H., Panda, A., Bhatu, P.: Conflvm: A compiler for enforcing data confidentiality in low-level code. EUROSYS (2019). doi:10.1145/3302424.3303952
34. Milburn, A., van der Kouwe, E., Giuffrida, C.: Mitigating information leakage vulnerabilities with type-based data isolation. IEEE SP, 1049–1065 (2022). doi:10.1109/SP46214.2022.00016. IEEE; IEEE Comp Soc
35. Neugschwandtner, M., Sorniotti, A., Kurmus, A.: Memory categorization: Separating attacker-controlled data. DIMVA **11543**, 263–287 (2019). doi:10.1007/978-3-030-22038-9_13. German Informat Soc, Special Interest Grp Secur Intrus Detect & Response; Trend Micro; Svenska Kraftnat; Recorded Future; Palo Alto Networks; Springer
36. van der Kouwe, E., Kroes, T., Ouwehand, C., Bos, H., Giuffrida, C.: Type-after-type: Practical and complete type-safe memory reuse. ACSAC, 17–27 (2018). doi:10.1145/3274694.3274705. Appl Comp Secur Associates; Natl Sci Fdn; U S Dept Homeland Secur Sci & Technol Directorate; Charles Koch Inst
37. Wang, G., Estrada, Z.J., Pham, C., Kalbarczyk, Z., Iyer, R.K.: Hypervisor introspection: A technique for evading passive virtual machine monitoring. USENIX WOOT (2015)