Computers & Security 120 (2022) 102781

Contents lists available at ScienceDirect

Computers & Security

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

KPointer: Keep the code pointers on the stack point to the right code

YongGang Li[a], Yeh-Ching Chung[b], Yu Bao[a], Yi Lu[a], ShanQing Guo[c], GuoYuan Lin[a,*]

[a] *The School of Computer Science and Technology in the China University of Mining and Technology, Xuzhou, Jiangsu, 221116, PR China*
[b] *The School of Data Science, CUHK(SZ), Shenzhen, Guangdong, 518172, PR China*
[c] *The School of Cyber Science and Technology, Shandong University, Jinan, Shandong, 250100, PR China*

ARTICLE INFO

Article history:
Received 5 November 2021
Revised 23 May 2022
Accepted 29 May 2022
Available online 2 June 2022

Keywords:
Integrity
System architectures
Security and protection
Software and system safety
Code reuse attacks

ABSTRACT

Affected by vulnerabilities, the control data on the stack is easily destroyed, which provides the most convenient conditions for code reuse attacks (CRAs). The operating system (OS) does not impose strict restrictions on the control flow paths. It allows instructions to jump to any location in the same address space. The OS will prevent code execution if and only if an execution error occurs. However, attackers can use stack overflow to accurately tamper with the control data on the stack and avoid execution errors. Although canary technology has been widely adopted, it turns out that this method can be bypassed. The traditional shadow stack technology can only protect the backward control flow and is invalid for the forward control flow. In contrast, the defense effect of the control flow integrity methods is better. Unfortunately, they either cannot get rid of the source code dependence on the protected objects, or cannot provide high-precision instruction boundaries. All these problems make it difficult to eliminate the CRAs based on stack overflow. Faced with these problems, this paper proposes a new security method KPointer. It filters the vulnerable data by tracking the overwriting operation to the stack data. Next, these data will be tracked to locate the jump instructions related to them. Finally, we use new security strategies to determine whether the current instruction is illegal. Experiments and analysis show that KPointer has a good protection effect on the CRAs based on stack overflow. It does not depend on the source code of the protected objects and only introduces 2.7% performance overhead to the CPU.

1. Introduction

In the OS, the stack is used to store local variables and specific information of the execution entities. Control data (such as return addresses and function pointers) are mixed with a large number of local variables on the stack. The out-of-bounds behavior generated by execution entities may damage the control data when the local variables on the stack are overwritten. CRAs (Guo et al., 2018; Yuan et al., 2015) use this feature to launch the most extensive control flow integrity (CFI) (Van der Veen and Andriesse, 2015) attacks on the OS. ROP (Return-Oriented Programming) (Payer and Barresi, 2015) and JOP (Jump-Oriented Programming) (Li et al., 2018) are the two basic attacks of CRAs, and other attack variants are derived from them.

The entire attack process of CRAs consists of three key steps: searching for gadgets, tampering with control data, and connecting gadgets. Tampering with control data is a necessary operation of CRAs. Currently, the vast majority of CRAs use overflow vulnerabilities to tamper with control data. Stack overflow vulnerabilities are the most common in all overflow vulnerabilities. As a result, the control data on the stack has become the preferred target of CRAs

To defend against CRAs, the direct method is completely eliminating gadgets (Fu et al., 2016). Unfortunately, it is almost impossible. There are many gadget forms, and new forms are constantly emerging. For example, the granularity of gadgets has changed from a code snippet containing only a few instructions to a complete function containing more instructions. Therefore, it is difficult to collect all gadget forms. In addition, the code has non-aligned characteristics, and different alignments will derive different code forms. For example, if *rip* points to the first byte of the binary code "*ff 25 ff e0 27 00*", the executed instruction is *jmpq *0* × *27e0ff (%rip)*; if *rip* points to the third byte, the executed instruction is *jmpq *%rax*. As a result, only disassembling the executable file cannot enumerate all gadgets. To make matters worse, different alignment combinations will produce massive amounts of code, which makes it extremely difficult to analyze and eliminate all gadgets.

Although gadgets cannot be completely eliminated, researchers can prevent attackers from gaining available gadgets. To achieve this goal, methods such as address randomization (Larsen and Franz, 2020), memory hiding (Fu et al., 2018) and code encryption (Qiu et al., 2016) were proposed. However, methods such as canary,

* Correspondence author.
E-mail address: liyg@cumt.edu.cn (G. Lin).

https://doi.org/10.1016/j.cose.2022.102781
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

stack address randomization and function pointer hiding have been defeated by memory leakage (Fu et al., 2018; Guo et al., 2018), and shadow stack can be bypassed by LOP (Lan et al., 2015). To make matters worse, attackers have developed code probing techniques, such as (Bittau and Belay, 2016; Gawlik et al., 2016; Lu and Song, 2015; Oikonomopoulos et al., 2016). No matter how to manipulate code, code probing can still collect enough information to construct available gadgets.

CFI does not care about diversified gadgets, nor does it worry about information leakage. It formulates the boundary of the jump instructions so that the control flow can only jump to a legal position. However, there are still serious challenges in defining the legal boundary. The analysis of Li (Li and Wang, 2020) shows that the existing CFI methods still cannot work out a perfect boundary. In practice, static CFIs have boundary accuracy problems, which leads to the risk of bypassing. High-precision dynamic CFIs are difficult to get rid of the dependence on the source code, which makes them invalid for the objects that do not contain source code (such as the loaded libraries). In addition, the target set of the indirect control transfer (ICT) instructions will increase dramatically as the amount of software code increases. This will increase the workload of the security tools, thereby affecting the OS performance.

Facts have proved that it is difficult for the existing general methods to have an ideal defense effect on CRAs based on stack overflow. The fundamental reason is that the general security methods do not fully consider the inherent relationship between the stack data and the code logic. We take the shadow stack as an example to illustrate the impact of ignoring such relationship. Arbitrary jump (Bittau and Belay, 2016) can make the control flow jump to an operand *c3* (may be part of the operand, but not an instruction), and make the operand become the opcode *ret* (machine code is also *c3*). Since the shadow stack mechanism does not set a detection point at an operand, the *c3* at this position does not trigger data verification when it is executed as an instruction *ret*. Therefore, the shadow stack mechanism can be bypassed successfully. In fact, *c3* at this position is essentially an operand rather than an opcode, and it has no relationship with any stack data. However, when it becomes an opcode, it can read the stack data to hijack the control flow.

In practice, the relationship between code and data depends on the execution logic of the code. Each piece of control data on the stack has a relationship with a specific instruction, and it cannot be used by other instructions other than the related one. Such relationship means that the specific control data can only be used by the specific instructions. Moreover, the specific stack data cannot be arbitrarily modified. For example, the return address cannot be modified before it is used by the instruction *ret*. The deployment of CRAs will inevitably break the relationship between the indirect control transfer (ICT) instructions and the control data. If we can find the relationship between code and data is broken, we can detect CRAs. µCFI (Hu et al., 2018) uses this method to construct a highly accurate ICT boundary, and thus has a strong defense against CFI attacks. However, it has a dependency on the source code, which makes it invalid for the loaded libraries.

If a security method can not only determine the relationship between control data and ICT instructions, but also get rid of the dependence on source code, it is an ideal method. Focusing on this goal, we propose KPointer to defend against CRAs based on stack overflow. Its main contributions are as follows:

(1) Establish a mechanism to identify vulnerable data. An identification mechanism is used to determine whether the overwritten stack data is vulnerable data in this paper.
(2) Establish a method to locate ICT instruction related with the vulnerable data. A multi-code space mechanism is built to locate ICT instructions. And a data back-tracing mechanism is proposed to find the transfer path of the vulnerable data.
(3) Formulate new security strategies to detect CRAs. Security policies that follow the rules of instruction execution and data update are built to defend against stack overflow-based CRAs.

2. Related works

Researchers have conducted extensive research on CFI protection, which mainly includes three categories: control data protection, jump target confusion, and CF path restriction. In this section, we describe these methods separately.

2.1. Control data protection

Due to lack of memory safety in C/C++, security vulnerabilities such as buffer overflows are frequently found (Sui and Ye, 2016; Szekeres and Payer, 2013; Ye and Su, 2014), which provides the possibility for attackers to destroy control data. Protecting control data can effectively mitigate CRAs. StackGuard (Cowan et al., 1998) fills a random value between the local variable and the return address, and verifies whether the value is changed when the function returns. But attackers can still bypass this method by exploiting some information disclosure vulnerabilities (Riq, Nov 1, 2021). Almost all shadow stack methods, such as (Fan and Sui, 2017), have the same problem. Similarly, Data hiding (Fu et al., 2018) and encryption (Qiu et al., 2016) have been defeated by memory probing technology (Bittau and Belay, 2016; Gawlik et al., 2016; Lu and Song, 2015; Oikonomopoulos et al., 2016).

Moreover, the most security methods are only valid for the control data with relatively fixed positions (such as return addresses). Especially the function pointers (local variables) stored on the stack, their storage locations are not fixed and may be recycled at any time. The existing methods can only find them by modifying the source code or performing compilation analysis, which leads to source code dependence.

2.2. Jump target confusion

The attacker needs to transfer control flow to the gadgets after tampering with control data. Obfuscating the jump targets can make it difficult to gain the gadget addresses. The current obfuscation methods mainly use address randomization (Marco-Gisbert and Ripoll, 2019).

CCFIR (Zhang et al., 2017) is a coarse-grained randomization method for binary code, which can be bypassed by the method in (LucasDavi et al., 2014). Marlin (Gupta et al., 2013) is a fine-grained randomization method, and it decomposes the binary file into multiple parts with functions as code blocks. Then all parts are mixed up. ILR (Hiser et al., 2012) even can randomize each instruction. The purpose of these methods is to prevent the attacker from knowing the address of gadgets. However, CLONE-ROP (Szekeres and Payer, 2013) can obtain the address by cloning the parent process. Although STABILIZER (Curtsinger and Berger, 2013) and RUNTIMEASLR (Kangjie et al., 2016) can use periodic or real-time randomization to overcome the process cloning problem, the high cost makes the both methods impractical. In addition, execute-only memory (XOM) (Kwon et al., 2019) also has a certain defensive effect on CRAs. But it doesn't work for arbitrary jmp[9].

The root cause of the flaws in these methods is the gadgets cannot be completely eliminated. Attackers can still obtain the gadget address by process cloning, information leaking, etc.

2.3. CF path limitation

The hijacked control flow will change the original execution paths. Researchers use the path restriction (Bounov et al., 2016;

Ge et al., 2016; Mashtizadeh et al., 2015; Niu and Tan, 2015; Tice and Roeder, 2014) to prevent ICT instructions from jumping outside the specified area. πCFI (Niu and Tan, 2015) is a fine-grained method based on MCFI (Niu and Tan, 2014). However, it requires operating source code and is invalid for the loaded objects (such as the loaded library). Similarly, although µCFI (Hu et al., 2018) has a good defensive effect, it cannot get rid of the dependence on the source code. Besides, there exist more coarse-grained CFI methods including KCoFI (Criswell and Dautenhahn, 2014), binCFI (Zhang and Sekar, 2013), and O-CFI (Mohan et al., 2015), etc. But the coarse-grained control flow graphs are too permissive so that they are still possible to mount attacks in general.

The ideal CFI methods need to determine the unique jump target of each ICT instruction when it is executed. However, due to different input and execution conditions, the jump targets are difficult to be determined. Moreover, the control flow graphs built by the existing methods may cause the status explosion problem. That is, too many redundant ICT targets have been covered in a huge set. On the one hand, these targets will increase the analysis burden, on the other hand, they may also introduce dangerous control flow paths.

3. Assumptions and threat model

First, we assume that attackers can use the overflow vulnerability to tamper with the control data on the stack. Second, we assume that the attacker cannot tamper with the code. In practice, the code segment will be mapped as non-writable. The attacker can only write the code by turning off the write protection of the code pages. Fortunately, we can use hardware virtualization technology to limit this operation, and we can also use EPT (Enhanced Page Table) technology to prevent all execution entities from writing the code segment. Third, we only consider the CRAs based on stack overflow. Other CFI attack forms are not considered in this paper. Fourth, we assume that all functions that can overwrite the stack data are known. In the known attack scenarios, CRAs use the functions that can manipulate memory or string to tamper with control data. Most of these functions are called in the form of library functions (such as *memcpy* and *fd_read*). We only need to analyze the corresponding library files to get them. Although some special functions or inline functions will be involved in the user code, we can still identify them through binary analysis. Moreover, DEP is enabled by default. To successfully deploy different types of CRAs, we turn off ASLR. In fact, ASLR does not affect the deployment of KPointer.

4. Overall design of KPointer

The control data on stack includes the return address (backward jump), the code pointer (forward jump), and the reference to the code pointer. However, the control data and non-control data on the stack are mixed together and their storage locations are not fixed, which brings challenges to the control data identification. Existing methods use compiler-based techniques to mark and analyze code pointers on the stack. But these methods cannot be used to protect the objects that do not contain source code, such as dynamic libraries. In scenarios where the source code is not available, we can't determine whether the data on the stack is control data. How to locate and identify the control data on the stack is the first challenge KPointer faces.

The update frequency of stack data is high, and it is difficult to predict. Tracking every piece of data in real time will inevitably introduce huge performance overhead. To make matters worse, it is difficult to analyze whether the loaded data is control data. In fact, only the data used as operands by instructions is control data. Even if we can know all the contents on the stack, it is difficult for

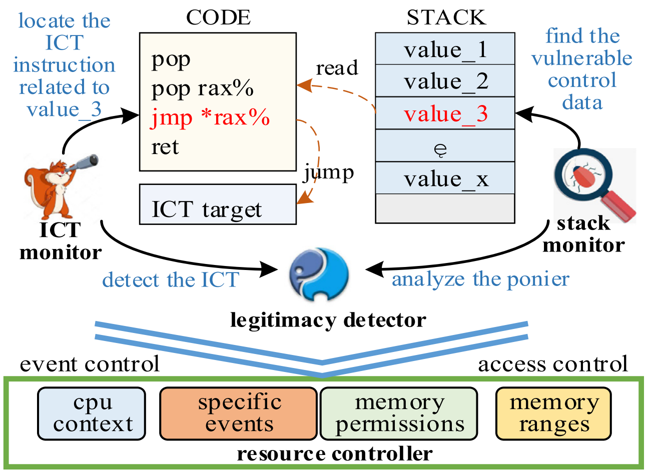


Fig. 1. The architecture of CFTS.

us to analyze which piece of data will be used as an operand. If we can know the relationship between the stack data and instructions, we can determine whether the data is used as an operand.

The existing security methods take all the ICT instructions as the detection targets. They have the problem of excessive redundancy targets. For the CRAs based on stack overflow, how to reduce the redundancy of the target objects is the second challenge KPointer faces. In a real attack scenario, only the ICT instructions related to the control data on stack will be used by CRAs based on stack overflow. Therefore, finding such instructions can reduce the redundancy.

After CRAs destroy the control data on the stack, the control flow will jump to a gadget chain. Traditionally, CFI method uses a high-precision boundary set to identify illegal jumps. However, it turns out it is difficult to formulate a high-precision boundary set for ICT instructions (Li and Wang, 2020) without the source code. How to formulate new security strategies without source code to achieve strong CFI protection is the third challenge that KPointer faces.

We find that CRAs will destroy the relationship between ICT instructions and control data. In addition, the illegal stack overwriting operations may also destroy the characteristics of the original data on the stack, and even reveal the attacker's overflow intention. In theory, the security strategies developed around these attack characteristics can detect CRAs.

According to the above analysis, KPointer needs to find the vulnerable control data on the stack. Then the relationship between the control data and ICT instructions should be identified. Finally, the relationship between control data and ICT instructions will be analyzed to determine whether CRAs exist. To achieve these goals, KPointer must have the ability to monitor and control the behavior of the executing entities. The overall design of KPointer is shown in Fig. 1.

KPointer is composed of *resource controller, stack monitor, ICT monitor* and *legitimacy detector. Resource controller* is responsible for monitoring and controlling the behavior of execution entities. It can control their resource access, which can provide basic deployment conditions for other components. *Stack monitor* is responsible for finding the vulnerable control data and setting it as the target object to be protected. Accurately identifying the vulnerable control data on the stack can reduce the number of target objects, thereby reducing meaningless protection operations. *ICT monitor* is responsible for locating ICT instructions that are related to vulnerable control data. CRAs will use such ICT instructions as gadgets, and the tampered control data is the operand of the instructions. In practice, not all ICT instructions can be used in CRAs. For example, in the code snippet "*mov 0 × 400880 %rax, jmp *%rax*", the ICT instruction *jmp *%rax* cannot jump to the next gadget. Because in this code snippet, the attacker cannot tamper with the operand of the ICT instruction. Only the ICT instructions that are related to vulnerable control data can be used as CRAs gadgets. The instructions that can be used by CRAs are the meaningful objects to be tracked and analyzed. The *legitimacy detector* is responsible for determining whether CRA is occurring in the OS. If a CRA is occurring, the *legitimacy detector* will immediately block the execution of the current execution entity.

5. The implementation of KPointer

In this chapter, we introduce the implementation of the four components of KPointer. The content includes: how to monitor and control the behavior of execution entities, how to find vulnerable control data on the stack, how to locate the ICT instructions related to the vulnerable control data, and how to determine whether CRAs are occurring in the OS.

5.1. Monitor and control the resource access

The behavior characteristics of all execution entities will be shown through the resources they access (such as memory, CPU, etc.), which cannot be hidden. By monitoring the accessed resources, we can know the state changes of the execution entity without the source code. In addition, by manipulating the resources of the target object, we can control the behavior of the execution entity.

However, the current OS does not provide the resource access control interfaces. Although the binary instrumentation does not depend on the source code, it requires manual intervention. In addition, binary instrumentation can only manipulate instructions and cannot manipulate memory (especially a large amount of memory). In response to these problems, we propose a resource access control model to control memory and specific events.

The VMX (virtual machine extensions) technology in modern CPUs can provide new execution modes for the OS, *VMX root* and *VMX non-root*. The model we proposed uses VMCS (virtual machine control structures) to develop a series of control strategies. Any operation that violates the strategies will cause the OS to switch from *VMX non-root* mode to *VMX root* mode (called system trap in this paper). In the *VMX root*, we can detect and modify the state of the execution entity to monitor and control its behavior. After the system trap event is over, the OS will return to *VMX non-root* again to continue execution. Below, we introduce the implementation of the control model in detail.

5.1.1. Monitor and control memory

This mechanism is used to adjust the memory permission and area dynamically, as shown in Fig. 2. EPT (Extended Page Tables) provides the basic ability to set memory permissions. Through the last three bits (*w*, *r*, and *x*) of the EPT's last-level page tables, we can control the memory permission with page granularity (4KB). Any memory access that violates the permission settings will be captured by KPointer.

To adjust the memory area that the target objects can access, KPointer introduces two methods: page redirection and EPT switching. Page redirection can make the same virtual address map to different physical memory by alternately rewriting EPT entries (*this_item* and *that_item* in Fig. 2). It can dynamically redirect each virtual page to any location in the entire physical memory. However, it will trigger a system trap. EPT switching uses the instruction *vmfunc* to switch the entire EPT without causing any system trap. Therefore, compared with page redirection, it is not only suitable for large-scale memory switching, but also introduces less

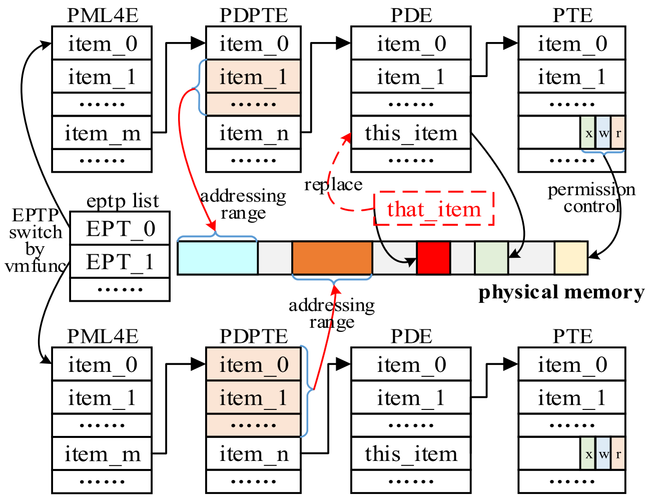


Fig. 2. Memory control mechanism.

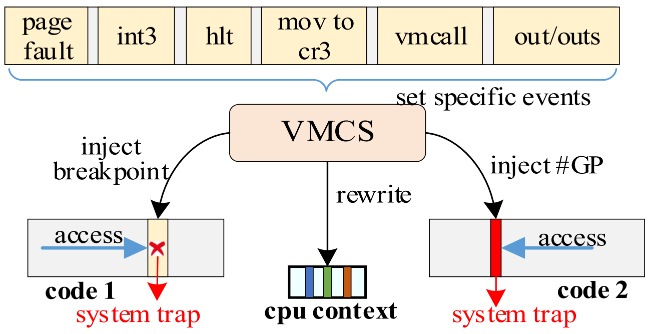


Fig. 3. Event control mechanism.

performance overhead. However, the number of available EPTs is limited by the capacity of the *EPTP list* to only 512.

5.1.2. Monitor and control the specific events

The event control mechanism can control specific events in the OS, as shown in Fig. 3. It can set breakpoints, inject general protection exceptions (*#GP*), rewrite CPU context, set interrupt exceptions, set process switching exceptions and inject system trap events (such as *int3* and *vmcall*).

We can enable up to 4 breakpoints (including data and instruction breakpoints) at the same time by setting the registers *dr0*~*dr3* and *dr6*~*dr7*. Then, the *bit 1* of *exception bitmap* in VMCS will be set to be *1*. When an execution entity accesses the breakpoints, it will trigger a system trap. The breakpoints can be used to track the execution entities with a single instruction or a single byte as the granularity.

If there is an operation that needs to be blocked immediately, a general protection exception will be injected into the OS by enabling the *vm-entry interruption* field in VMCS. Then, the next execution will trigger a system trap, and the subsequent operations of the execution entity will be terminated.

The action of the execution entity can be controlled by rewriting the CPU context in VMCS. For example, rewriting the *rip* register in VMCS can control the execution paths, and modifying the *rbp* and *rsp* can change the stack frame of the function. Moreover, by setting the *TF* bit of the *eflag* register and the *BS* bit of the *dr6* register, the single-step debugging mode is enabled. After that, the OS will trigger a system trap after each instruction. Then, we can monitor the execution of each instruction. For other events to be monitored, such as *in3*, we just need to set the corresponding VMCS fields.

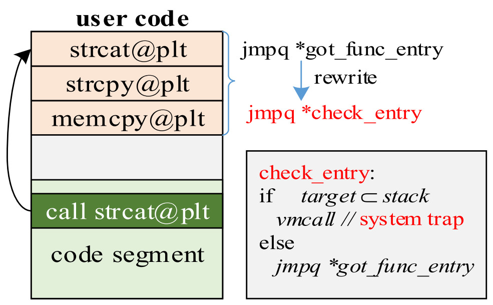


Fig. 4. Stack overwrite detection method.

5.2. Find the vulnerable control data on the stack

Stack overflows are caused by memory or string manipulation functions and they lacate in dynamic libraries. For example, the library function *fd_read* in *wget* can be used to overwrite the return address on the stack (*CVE-2017-13089*). In theory, monitoring such library functions can detect potential attacks. However, it's difficult to judge the legitimacy of these functions at binary level. Because the memory overwritten by these functions is not fixed, and it's difficult to predict whether the overwritten content is control data. The key to solving this problem is we can identify whether there exists control data overwritten by these functions on the stack.

In practice, not all control data on the stack can be attacked. First of all, whether it is the return address or the function pointer, an attacker needs to use a stack overflow to tamper with it. In addition, the control data must have been assigned before being tampered, and the tampered control data cannot be assigned again before being read. Otherwise, the execution entity will restore the tampered control data to a legal value according to the original execution logic, which loses the meaning of tampering with the control data. In general, only those control data that can be changed by the overwriting function are vulnerable control data.

According to the above analysis, the vulnerable control data on the stack has two basic characteristics: it can be changed by an overwriting function, and it is a piece of address data pointing to code. To identify such data, we must be able to detect whether the overwriting function is writing address data to the stack. The detection method is shown in Fig. 4.

Generally, execution entities need to use specific library functions to overwrite stack data. Modern code is position-independent code (PIC). The execution entity needs to use PLT (Procedure Linkage Table) and GOT (Global Offset Table) to call library functions, such as *memcpy*. Therefore, we can detect the intention that the execution entity overwrites the stack by monitoring the PLT and GOT.

Our targets are the functions with stack overwrite capability, including *strcat, strcpy, memcpy, fscanf*, which is shown as Appendix B. The first instruction *jmpq *got_func_entry* in PLT entries calling these functions will be rewritten as *jmpq *check_entry*. When these functions are called, the control flow will jump to *check_entry* instead of the library function. *Check_entry* will verify whether these functions attempt to write data to the stack. If yes, *vmcall* will be executed to trigger a system trap; if not, the original instruction *jmpq *got_func_entry* will be executed to make the control flow follow as the original path. This method can detect whether the execution entity attempt to overwrite the stack data.

Moreover, all "*rep xx xx xx*" and LOOP code blocks are also potential targets to be monitored. By detecting whether the code block can write data whose size is non-fixed into the stack, it can be determined whether it is a target. For example, if "*%es:(%rdi)*" in the instruction "*rep stos %rax, %es:(%rdi)*" points to the stack, and the number of bytes written is determined by *rcx*, it should be monitored. We rewrite these instructions and redirect them into check code. The check code can automatically verify whether they are continuously writing data to the stack. In short, if a code block can continuously overwrite the stack and the number of bytes written is not fixed, it is a code block that can continuously overwrite stack data, and it will be monitored.

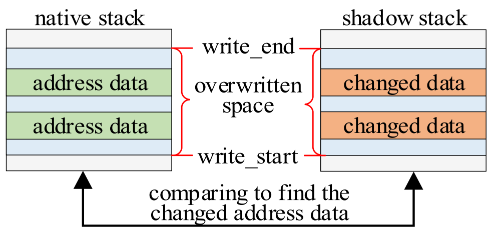


Fig. 5. The detection method of control data.

After the overwriting intention is detected, we check whether there is address data on stack. The detection method is shown in Fig. 5. After *check_entry* triggers a system trap, a shadow stack will be created for the current execution entity. We can switch the current native stack to shadow stack by rewriting the *guest rsp* and *guest rbp* fields in VMCS. After that, the overwriting functions will write data to the shadow stack. Finally, the changed address data (called changed data) can be detected by comparing the native stack and shadow stack. The target address data we are concerned about has two characteristics. One is the data directly or indirectly points to the code before being overwritten, and the other is it still directly or indirectly points to the code after being overwritten. That is, they are code pointers or references to the memory containing code pointers. Such address data may be vulnerable control data and is called suspicious data in this paper. How to determine whether these data are control data will be introduced in the next section.

5.3. Locate ICT instructions related to control data

The suspicious data found in Section 5.2 is not necessarily vulnerable control data. Even these address data may not be control data. The non-control data is not the operand of the ICT instruction, and it cannot be used by CRAs. CRAs can be successfully deployed if and only if the tampered data is used as the operand of the ICT instruction. If an ICT instruction uses the suspicious data found in Section 5.2 as an operand, it is an instruction related to vulnerable control data. And the suspicious data is the vulnerable control data. Therefore, the key to locating such an instruction is to detect whether any suspicious data is used as a jump target by an ICT instruction.

After identifying the suspicious data, we can track each instruction, and we can also track the transfer path of each data to detect whether the suspicious data is used by ICT instructions. However, this method introduces a lot of redundant operations. In real attack scenario, the tampered control data will be read and passed to the ICT instruction. Only suspicious data that has been read can be used by CRAs. Therefore, we only need to track the suspicious data that has been read. At the same time, we take the read operation of the suspicious data as the trigger condition for tracking ICT instructions. This method can reduce redundant operations and improve detection efficiency.

5.3.1. Detect the suspicious data reading

To detect the suspicious data reading operation, we need to set the suspicious data as unreadable. There are 4 debug registers (*dr0* ∼ *dr3*) in a CPU core that can be used to set reading breakpoints. When the suspicious data is less than or equal to 4, we use the debug registers to set them as unreadable. When the suspicious

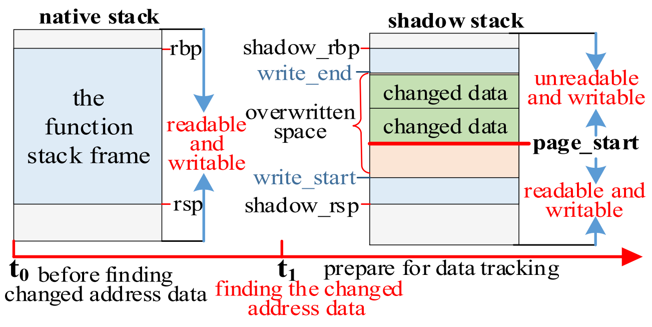


Fig. 6. The stack offset mechanism.

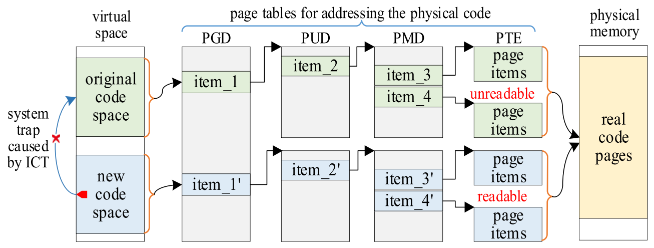


Fig. 7. The locating mechanism of ICT instructions.

data exceeds 4 pieces, we use the stack offset mechanism to put all the suspicious data in an unreadable page(s), as shown in Fig. 6.

In the stack offset mechanism, we move the overwritten data in the shadow stack to the memory head, until the first suspicious data is at the head of the memory page. In this way, all the suspicious data and the current content pointed to by *rsp* are located in different pages. The page(s) where the suspicious data is located will be set as unreadable. Therefore, any suspicious data reading will trigger a system trap. It should be noted that in addition to suspicious data, there are original data in unreadable page(s). And the original data reading will also trigger a system trap. We will enable these operations to obtain their target data through single-step debugging.

Compared with the breakpoint method, the stack offset mechanism is less efficient. The stack offset mechanism will be activated when and only when there are more than 4 pieces of control data overwritten on the stack. Fortunately, we did not find a situation that triggers the stack offset mechanism in our experiments. We have analyzed a large number of source code including *libc*, *lmbench*, *nbench*, and *speccpu2006*, and found none of them overwrite more than 4 pieces of control data on the stack. Therefore, we believe that the stack offset mechanism only exists in very rare situations or attack scenarios, and its activation frequency will be very small.

5.3.2. Locating ICT instructions related to suspicious data

Next, we check whether the suspicious data being read is used as the operand by an ICT instruction. The suspicious data being read may not be control data. Therefore, we must determine whether it is control data. For control data, we also need to track its transfer path to locate the ICT instruction related to it. Suspicious data may be directly used as the operand of an ICT instruction, or it may be used after being changed. The changed control data is not the same as the original data, which brings challenges to suspicious data tracking.

To track suspicious data and locate the related ICT instructions, we establish a location mechanism, as shown in Fig. 7. We create a new code space for the execution entity, its size is the same as the original code space. The two code spaces use different addressing page tables, and both point to the same physical code. For example, if the original code space range is $0 \times 400000 \sim 0 \times 409000$, the new code space may be $0 \times 68ff400000 \sim 0 \times 68ff409000$. Next, the last page table PTE of the original code will be set as unreadable (set by EPT). Finally, the *guest rip* in VMCS will be rewritten to make it point to the new code space. Therefore, the control flow will flow in the new code space instead of the original code space.

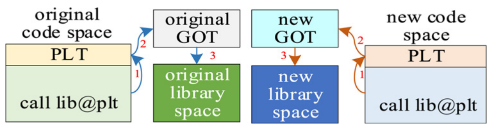


Fig. 8. New GOT space.

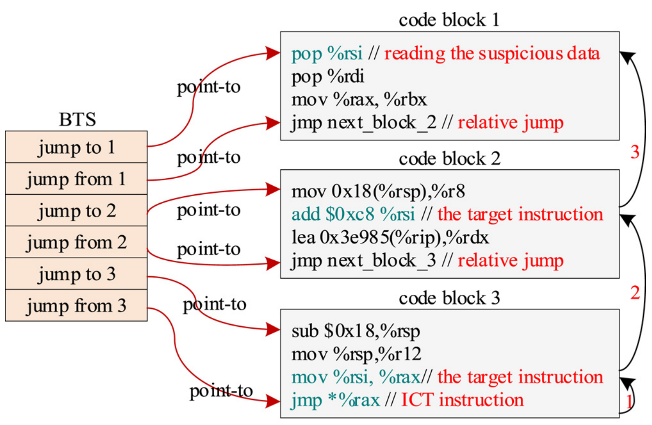
In the new code space, all instructions with consecutive addresses and all relative jump instructions can be successfully executed. These instructions are based on *rip* to calculate the next instruction. So, their next instructions are still in the new code space. And they can accurately locate the real physical code. In contrast, all ICT instructions will trigger a system trap and be captured by KPointer. The reason is the ICT instructions will jump to the original code space, and the PTE in this space is unreadable. As a result, ICT instructions will trigger system traps due to accessing the unreadable PTE.

Considering some code is shared, we also need to ensure other execution entities can call the shared code in the original space. In user space, multiple threads share the same application code and library code, and different processes share all library code. For processes, our method does not affect their library function calls. Because we only change the PTE permissions of a certain process, not the permissions of the entire library. The page tables used by different processes are not the same. Therefore, library function calls of other processes will not trigger any system trap due to PTE.

However, the new code space will cause call errors when using PLT. Because PLT uses the address in *rip* as the base address to calculate the GOT address. After adding a relative offset, the new address will point to entries in the GOT. But the current *rip* has been changed to point to new code space, which makes PLT unable to find the original GOT. To solve this problem, we established a new GOT for the new code space, as shown in Fig. 8. The new GOT was exactly the same as the original GOT when it was first created. After that, all entries that point to the code will be modified (plus the offset between the two code spaces) to make it point to the new library space. Therefore, the code *call lib@plt* in the new code space can also jump to the library function through the new GOT.

For threads, they share the same page tables. The PTE of the original code space is unreadable, which will cause other threads to be unable to access the original code space. To solve this problem, we need to enable threads other than the target thread to normally call the code in the original code space. We create a new EPT for the target thread, and other threads use the original EPT. This method can be achieved by rewriting the *EPTP* field in the VMCS corresponding to the specific CPU core. The PTE of the original code space is unreadable in the new EPT, while it is readable in the original EPT. Therefore, the target thread jumping to the original space will trigger a system trap, but other threads will not.

After locating the ICT instruction, we need to determine whether it is related to suspicious data. If the ICT instruction takes suspicious data as an operand, the data needs to be transferred to the ICT instruction. The entire transfer process starts with the instruction reading data and ends with the ICT instruction. For example, the instructions "*pop %rax; pop %rbx; jmp *%rax*" can transfer suspicious data on the stack to the ICT instruction "*jmp *%rax*" through *rax*. We trace all instructions related to the operand of the ICT instruction. The tracing path is opposite to the order of the executed instructions. The whole process starts with ICT instruction



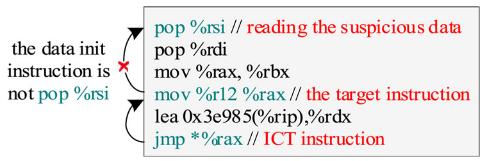
```
code block 1
pop %rsi // reading the suspicious data
pop %rdi
mov %rax, %rbx
jmp next_block_2 // relative jump

code block 2
mov 0x18(%rsp),%r8
add $0xc8 %rsi // the target instruction
lea 0x3e985(%rip),%rdx
jmp next_block_3 // relative jump

code block 3
sub $0x18,%rsp
mov %rsp,%r12
mov %rsi, %rax// the target instruction
jmp *%rax // ICT instruction
```

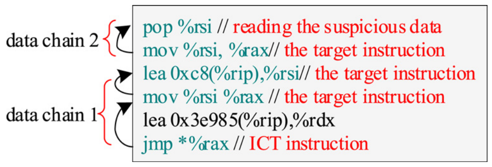
1. analyze the ICT instruction (jmp *%rax) to get the target register (rax) that is the instruction operand.
2. trace back to get the instructions (mov %rsi, %rax) that transfers value to the target register and record the source register (rsi); then, trace back to get the instructions (add $0xc8 %rsi) that related to the source register.
3. repeat step 2 until the instruction to read suspicious data is traced back.

Fig. 9. The reverse tracking of suspicious data.



```
pop %rsi // reading the suspicious data
pop %rdi
mov %rax, %rbx
mov %r12 %rax // the target instruction
lea 0x3e985(%rip),%rdx
jmp *%rax // ICT instruction
```

Fig 10. Data chain violating condition 1.



```
pop %rsi // reading the suspicious data
mov %rsi, %rax// the target instruction
lea 0xc8(%rip),%rsi// the target instruction
mov %rsi %rax // the target instruction
lea 0x3e985(%rip),%rdx
jmp *%rax // ICT instruction
```

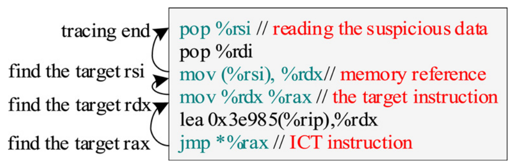
Fig. 11. Data chains violating condition 2.

and ends with the instruction reading data. If these instructions can form a complete data transfer chain, the current ICT instruction is related to the suspicious data. The tracing process of suspicious data is shown in Fig. 9.

The relative jump instructions in the new code space can continue to execute without triggering a system trap. These instructions divide all the executed instructions into several discontinuous code blocks. The transfer path of suspicious data is in these blocks. To collect these instruction blocks, we use Intel BTS (Branch Trace Store, enabled by setting the *bit 3* of *MSR_DEBUGCTL*) to record jump instructions.

To determine that the ICT instruction and the suspicious data are related, two strict conditions need to be met. First, the entire data transfer chain must start with the instruction reading suspicious data and end with the ICT instruction. Second, there is one and only one data transfer chain in the entire transfer process. The two data transfer processes in Figs. 10 and 11 violate the above two conditions respectively.

Both the ICT instruction locating method and the data tracing method designed in this paper have strong anti-interference. Any ICT instruction can be captured as long as it attempts to jump to the original code space. At the same time, the new code space is random, and it is difficult to probe. As for the suspicious data, even if it is changed many times during transfer process, it can still be traced back. The data tracing method can even trace the attacker's reference to the memory containing function pointers (ie, indirect pointers), as shown in Fig. 12. Because it focuses on the correlation



```
pop %rsi // reading the suspicious data
pop %rdi
mov (%rsi), %rdx// memory reference
mov %rdx %rax // the target instruction
lea 0x3e985(%rip),%rdx
jmp *%rax // ICT instruction
```

Fig. 12. Tracing memory reference.

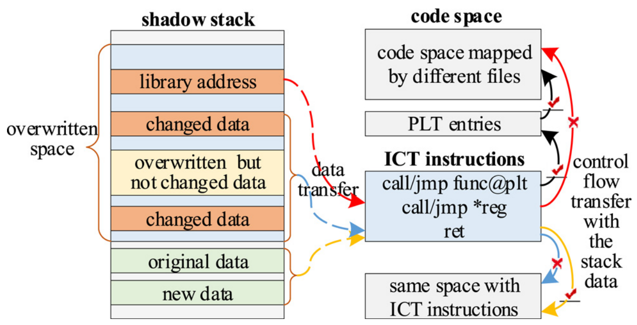


Fig. 13. The legitimacy judgment of ICT instructions.

between data transfer instructions. No matter what kind of attack method it is, it cannot eliminate the correlation between instructions.

When the address pointed to by the *rsp* register exceeds the location of the suspicious data, and the CPU context does not contain any suspicious data, the current tracking will be terminated. In addition, all suspicious data under the location pointed to by the current *rsp* will be destroyed to prevent them from being used maliciously. Every time an ICT instruction is detected, we will determine whether the conditions for terminating the tracking process are met at this moment.

It should be noted that the events such as system call, interrupt, and process switching during the tracking process will not affect the normal execution of the OS. Because we do not restrict the code permissions of the kernel and other execution entities. For the legal long jump, we will filter them out by searching *setjmp* and *longjmp* and allow them to jump to the target position.

5.4. Determining the legitimacy of ICT instructions

After locating the ICT instruction related to the suspicious data, we need to determine whether it is legal. The legitimacy judgment method is shown in Fig. 13.

For the backward jump instruction *ret*, we do not allow it to use any overwritten data as the return target. In practice, the return address will not be changed in any way before it is used by *ret*. Therefore, if *ret* uses suspicious data as the return address, it's illegal. Moreover, if the return address is rewritten but not changed, it also cannot be used by *ret*.

The data that the attacker attempts to tamper with may not be in the current function stack frame. Another word, the target data to be tampered is located at the upper memory of the return address. Therefore, the attacker needs to go over the return address to tamper with the target data. The OS adds a canary before the return address to protect it. The attacker needs to keep the canary unchanged. Besides, an inappropriate return address will cause an exception leading to process crash. Therefore, the attacker has to keep the return address unchanged. Attackers can use memory probing (such as BROP (Bittau and Belay, 2016) and CROP (Gawlik et al., 2016)) to gain the canary and return address. During overwriting the stack data, the attacker can bypass the stack protection mechanism as long as it rewrites the two original values to the original location on the stack.

Fortunately, we can find the canary and return address have been rewritten through the shadow stack mechanism. When *ret* uses a return address that has been rewritten but not changed, we judge it as illegal. Moreover, all the overwritten data will be judged as illegal. This design can limit the attacker's overwriting range to a function stack frame on the stack.

For the forward jump instructions that call library functions, all operations that jump to library functions through PLT are legal. If the instruction *call/jmp *register* uses suspicious data as an operand to make control flow jump to other libraries, it will be judged as illegal.

All library functions in Linux are PIC. The user does not need to know the location of the library, and he can reach the target function through PLT and GOT. In practice, all library files will be mapped into a series of discontinuous virtual spaces. The code must use PLT and GOT to jump between different mapping spaces. In contrast, CRAs will directly jump to the library code without passing PLT and GOT. Based on this difference, we can determine whether the library function is legally called. This design restricts the gadgets that can be used by attackers to the mapping space of one executable file.

In summary, the above method puts three restrictions on CRAs: only forward jump instructions can be used; only control data in a single function stack frame can be tampered with; only gadgets in the mapping space of a single executable file can be used. These restrictions can defend most CRAs.

However, there are still some clever CRAs that can be deployed. To improve the defense effect, we formulate some new security strategies, which are shown as the flowing.

First, for the control data itself, if it originally points to the head of the function, it will not point to the inside of the function after being overwritten; if it originally points to the inside of the function, then it will not point to the head of the function after being overwritten.

Second, for ICT instructions related to target control data, "*jmp **" only allows control flow jumping to the inside of the current function, while "*call **" can only jump to the head of the function. It should be noted that "*longjmp*" can be gained by parsing the "*longjmp()*" function in the ELF file, and we allow it to jump to the target address.

Third, stack data that is continuously overwritten should have the same properties except for the data structures and classes stored on the stack. For example, the overwritten strings are all character data, and function pointers do not appear in them.

Fourth, for data structures and classes, their member variables should maintain the same properties before and after they are overwritten. In addition, we found function pointers inside them are rarely overwritten continuously after being assigned. Therefore, if the function pointer is continuously overwritten, and other non-address data is changed at the same time, the operation will be judged to be illegal.

Fifth, control data does not become non-control data after being updated, and non-control data does not become control data after being overwritten.

In summary, the attack behavior in the same ELF file is also subject to multiple conditions. All control flow transfers that violate the above policy are illegal, and the previous stack overwrite operation is also illegal.

6. Evaluation

We conduct all experiments on a Dell T440 server, which is equipped with two 10-core Intel Xeon silver 4210 2.2 GHz CPUs and 128GB memory. The OS is Ubuntu-16.04 with kernel 4.15.0.

6.1. Security Analysis

To evaluate KPointer's defense effect on back-forward attacks, RIPE (Wilander et al., 2011) test suite is implemented. It consists of 850 buffer-overflow attacks that can tamper with the return addresses. We use RIPE to attack the OS 480 times, and test KPointer's defense action. The test results are shown in Table 1. We find the native OS has a certain defense effect on back-forward attacks caused by stack overflow. But it can still be bypassed by about 8% of attacks. In contrast, KPointer can prevent all back-forward attacks.

KPointer has strong protection effect on backward instructions. It does not indirectly mark the return address like StackGuard (Cowan et al., 1998), nor does it hide the code pointers like CodeArmor (Burow et al., 2017). Instead, KPointer verifies the legitimacy of the backward jump by tracing the changes of the control data on which the instruction *ret* depends. As long as the return address has been manipulated, even if the value of the return address is not changed, KPointer can detect such abnormal behavior. Therefore, whether it is canary probing or memory leak, KPointer cannot be bypassed.

To verify the effect of KPointer on forward control flow protection, we test four real-word applications (*Nginx, Proftpd, Mcrypt* and *TORQUE*) containing stack overflow vulnerabilities. To simulate the JOP attack, we add a null function *null_call()* to the source code of these four applications (*ngx_http_request_body.c* of *Nginx, netio.c* of *Proftpd, extra.c* of *Mcrypt*, and *disrsi_.c* of *TORQUE*). After that, we call *null_call()* through a function pointer (local variable) in the function containing a vulnerability. We deploy the compiled application in the OS to simulate a no-source execution scenario. Finally, we use 4 JOPs based on stack overflow vulnerabilities to attack the above programs, and check the defense effects of the security methods.

Attack 1 targets the web server *Nginx*. It constructs the JOP chain based on the existing knowledge of *Nginx* and *libc*, and exploits the stack buffer overflow vulnerability (CVE-2013-2028) to tamper with the pointer of *null_call*.

Attack 2 targets the ftp server *Proftpd*. It first locates gadgets by scanning the *Proftpd* executable and *libc*. Then, it reads the load addresses of *libc* from */proc/pid/maps* to determine the absolute address of gadgets. Finally, it sends the buffer containing the JOP chain to *Proftpd* via an unauthorized FTP link, which will replace the pointer of *null_call* on stack through the vulnerability (CVE-2010-4221) in *Proftpd*.

Attack 3 targets *Mcrypt*. It first obtains the load addresses of *Mcrypt* and *libc* from the */proc/pid/maps* to construct the JOP chain. It then sends the JOP chain to *Mcrypt* through a pipe. Next, it executes the JOP chain through tampering with the pointer of *null_call,* which is achieved by exploiting a stack buffer overflow vulnerability (CVE-2012-4409) in *Mcrypt*.

Attack 4 targets the *TORQUE* resource manager server. It first reads the load address of the *pbs_server* and constructs a JOP chain. It then sends the JOP chain to *TORQUE* through an unauthorized network connection, and exploits the stack buffer overflow vulnerability (CVE-2014-0749) in TORQUE to modify the pointer of *null_call*.

The results show that KPointer can detect and block these attacks. In contrast, MCFI (Niu and Tan, 2014), πCFI (Niu and Tan, 2015), CFI-LB (Khandaker et al., 2019), OS-CFI (Khandaker and Liu, 2019), µCFI (Hu et al., 2018) and PARTS (Liljestrand et al., 2019) failed to detect any of the above attacks. Because, these methods either rely on source code, or do not support the protection of shared libraries.

To compare the defense effect of KPointer with the similar methods, we analyze their defense effect against stack overflow-based CRAs. The results are shown in Table 2.

Table 1
Real attack defense.

Methods	total attacks	successful attacks	partly attacks	failed attacks
Native OS	480	37(7.7%)	2(0.4%)	441(91.9%)
KPointer	480	0	0	480(100%)

Table 2
The defensive effects of security methods against stack overflow-based CRAs.√: success protection, ×:failed to protect, ⋆:partial success protection.

Attack types	MCFI	πCFI	CFI-LB	OS-CFI	μCFI	PARTS	KPointer
code pointer overwrite	⋆	⋆	⋆	√	√	√	√
return address overwrite	√	√	×	×	√	√	√
tail call attack	⋆	⋆	×	×	√	√	√
vatbale injection	⋆	⋆	⋆	√	√	×	√
setjmp/longjmp	√	√	×	×	×	×	√
function type confusion	√	√	√	×	×	×	×
shared library gadgets	×	×	×	×	×	×	√

Table 3
Static statistics of instructions and data related to control flow transfer.

APP	call *	jmp *	ret	FPoS	RA	FPoH	FPoD
Nginx	309	33	1276	244	1276	124	975
Redis	742	682	6368	14	6368	1	3838
Httpd	1222	259	6928	48	6928	33	84

The results show that KPointer has better defense effect on stack overflow-based CRAs. However, it doesn't detect function type confusion. Because KPointer's defense targets are executable entities without source code. They have lost high-level semantics (including function types, parameter types, variable types, etc.). As a result, KPointer has no effect on function type confusion. It should be noted that once the control data used in the above attack is not on the stack, KPointer will lose its effect.

We count the number of control flow transfer instructions and control data in real applications, as shown in Table 3. FPoS: function pointers on stack; RA: return addresses; FPoH: function pointers on heap; FPoD: function pointers on data segment. The statistics show that the amount of control data on the stack (FPos and RA) is the largest. In a real execution scenario, whether it is ROP (Payer and Barresi, 2015), JOP (Li et al., 2018), LOP (Lan et al., 2015), JIT-ROP (Ahmed and Xiao, 2020) or any other CRA variant, it will be detected by KPointer as long as it needs to tamper with the control data on the stack.

However, KPointer cannot protect control data (less than 50%) on the heap and data segments. According to our observation, most of the control data in the heap and data segment exist in the form of function pointers, which all point to the head of functions. In addition, control data in the heap and data segments resides in memory longer and changes less frequently than control data on the stack. These features are beneficial for us to build conservation models.

6.2. Performance analysis

We use *SpecCPU2006* to test the performance loss of CPU introduced by KPointer, as shown in Fig. 14. The results show that the average performance loss is 2.7%. During the test, we found that the system trap frequency is a key factor affecting the performance. The test program will be suspended when a system trap is triggered. KPointer will take over the control flow until the trap event is over. In the whole process, the test program will lose a short period of execution time, causing its execution time to become longer.

SpecCPU2006 focuses on testing the CPU performance, and cannot test other performances well. To make up for this shortcoming, we use other applications as benchmarks, which is shown as Fig. 15.

Apache, Lighttpd, and Nginx focus on network performance. Redis and Memtester focus on memory performance. Gzip, Tar, and FIO focus on IO performance. All test items are measured against their respective runtimes when KPointer is not running. The results show that the performance loss of each test item is less than 8%, and the average performance loss of all test items is 4.2%.

The above applications do not generate too much suspicious data during the test. Therefore, we cannot observe the performance of KPointer when it tracks suspicious data and ICT instructions. To solve this problem, we modified RIPE (Wilander et al., 2011) to make it generate suspicious data. Two function pointers are designed, and they point to an empty function respectively. Before calling the overwriting functions (such as *memcpy* and *strcpy*), one pointer is stored on the stack. After that, the overwriting functions are called to write data to the stack, and the function pointer will be changed to another function pointer by the overwriting functions. In fact, neither of the two empty functions will be called. Otherwise, KPointer will judge it as an illegal operation and kill the current process, which will prevent us from continuing to observe the performance of KPointer tracking ICT instructions. It should be noted that we put the function pointer at 32 bytes from the overwriting start address. Therefore, when the number of the overwritten bytes exceeds 32, RIPE will generate a piece of suspicious data.

During the test, the number of bytes written to the stack will gradually increase. We record the running speed of the target functions and RIPE before and after the KPointer deployment. The experiment results are shown in Figs. 16 and 17. The *0* in abscissa means that the function writes data to the other memory instead of the stack.

In Fig. 16, when the overwriting functions do not write data to the stack, the effect of KPointer on them is not obvious. When they write a small amount of data to the stack, KPointer will have a significant impact on them. Because when the overwriting functions write data to the stack, KPointer has to create a shadow stack for them. The system trap, stack creation, and stack migration caused by this operation will affect the execution speed of the functions.

As the number of the overwritten bytes increases, the impact of KPointer on the overwriting functions will become smaller. Because the more bytes are written, the longer the execution time of the functions. The time for KPointer to build the shadow stack is relatively fixed, about 2μs. Therefore, the longer the running time of the function, the smaller the relative impact of KPointer.

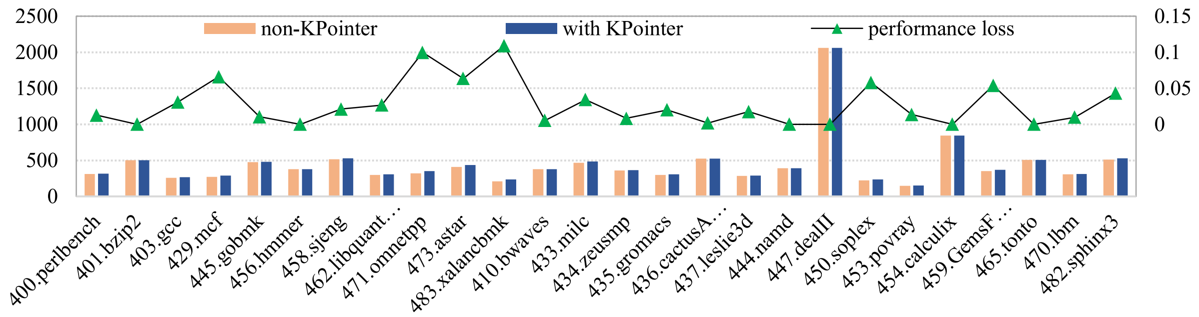


Fig. 14. Speccpu test results. The abscissa is the test pro.gram. The ordinate on the left is "base run time", which corresponds to the bar graph; the ordinate on the right is the performance degradation factor, which corresponds to the line graph.

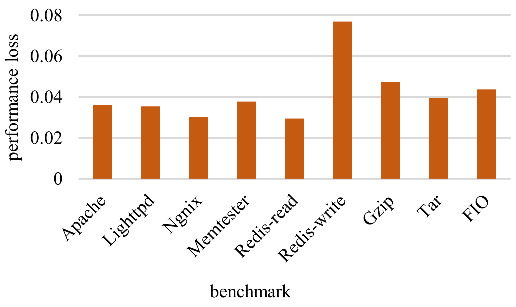


Fig. 15. The performance loss of real applications.

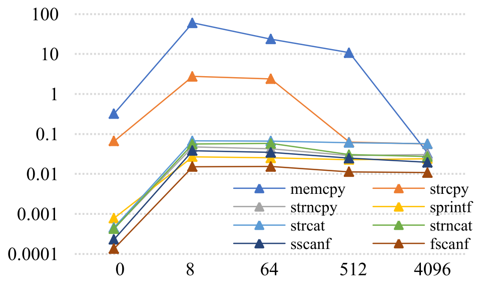


Fig. 16. The running speed attenuation of the overwriting functions. The abscissa indicates the number of bytes written to the stack. The ordinate represents the speed attenuation factor of each function after deploying KPointer.

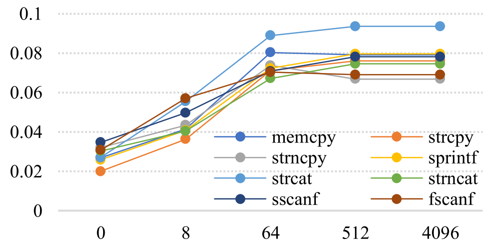


Fig. 17. The running speed attenuation of RIPE. The abscissa indicates the number of bytes written to the stack. The ordinate represents the impact of the overwriting functions on RIPE after the deployment of KPointer.

It should be noted that the speed attenuation factor of *memcpy* is extremely large at the beginning, and then rapidly decreases. Because when the number of the copied bytes is small, the library function *memcpy* will be replaced by the instruction *rep stos xx xx*. The running speed of the *rep stos xx xx* is much higher than the library function *memcpy*. The time that KPointer builds a shadow stack is dozens of times the execution time of the instruction *rep stos xx xx*. When *memcpy* copies more bytes, the library function is enabled. Then, the execution time of *memcpy* is increasing rapidly, which makes the relative impact of KPointer smaller.

Fig. 17 shows that the impact of KPointer on RIPE will increase as the number of bytes written into the stack increases. When the overwriting functions does not write data to the stack, the impact of KPointer on RIPE is between 2% and 3.5%. When data is written but no suspicious data is generated, the impact increases to 4%~6%. At this point, KPointer needs to create a shadow stack for the current execution entity. In addition, KPointer has to create a new code space and migrate control flow to the new space. These operations will slow down RIPE. When the number of bytes written into the stack reaches 64, the running speed of RIPE continues to slow down. Because at this time the process has generated suspicious data (located at the 32nd byte). Then, each ICT instruction will trigger a system trap. Fortunately, the impact of KPointer will not increase too much, and it will stabilize gradually.

To further test the performance overhead of the OS caused by KPointer, we introduce some microbenchmarks. The experiment results are shown in Table 4. We found KPointer does not significantly affect the execution speed of instructions other than the target ICT instructions. In contrast, it has a greater impact on ICT instructions related to suspicious data. Because the target ICT instruction triggers a system trap when it is executed, the processing time of a system trap exceeds 500ns. Next, KPointer determines the legitimacy of the ICT instruction based on the security strategies. The whole process requires about 3.3μs, which is far more than the execution time of an ICT instruction. Moreover, the memory exception caused by the EPT permission settings will also affect the running speed of the process. In short, all the time-consuming operations are caused by system traps.

Based on the above experiments, we can draw the conclusion that the system trap is the main factor that KPointer introduces performance overhead. The system traps generated by KPointer include unconditional traps and conditional traps.

Unconditional traps are caused by the specific instructions. These instructions include *CPUID, GETTSEC, INVD, XSETBV* and all VMX instructions except *VMFUNC*. The overhead they introduce is positively related to the number and frequency that the program calls these instructions.

The conditional traps are triggered by the security strategies set by KPointer, including memory permission exceptions and specific events (such as executing *int3*). The performance overhead depends on how often the system traps occur. For example, if the instructions *jmp *register* appear frequently after the suspicious data is detected, they will introduce a large performance overhead due to the frequent system traps.

In addition to affecting the performance of the OS and applications, KPointer also occupies some memory. These memories mainly include KPointer's code segment, data segment and EPT page table (switched by vmfunc). The first two are fixed and total less than 4MB. The latter is related to the memory size of the OS. In this paper, indexing 128GB of memory requires about 257MB of

Table 4
Micro benchmarks (Nanoseconds).

No KPointer					With KPointer (After overwriting the control data on the stack)						
call addr	library call	ret	jmp addr	jmp/call *reg	call addr	library call	ret	jump addr	system trap	ept exception	jmp/call *reg
2.78	3.09	2.69	1.37	1.39	2.78	3.1	2.71	1.37	579.61	1109.79	3271.83

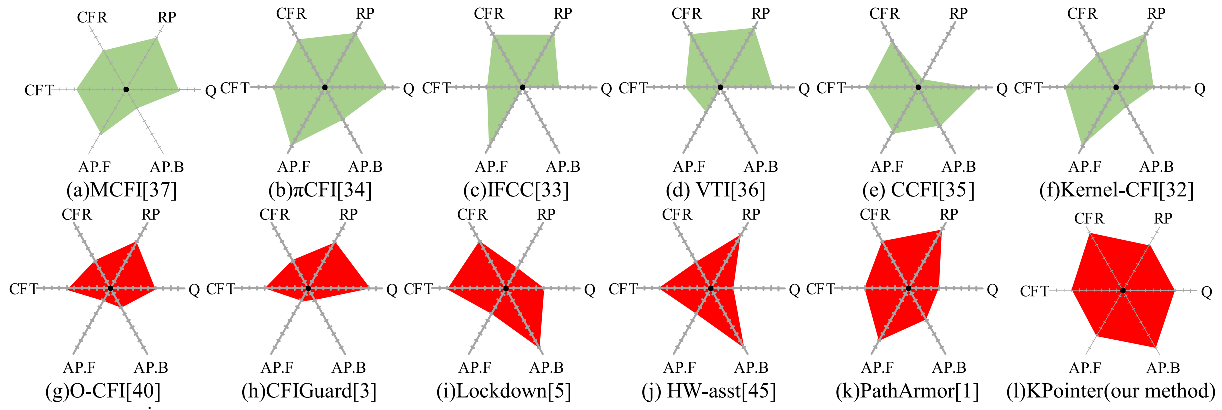


Fig. 18. Comparison with existing methods. Binary-based methods are red, source-based methods are green.

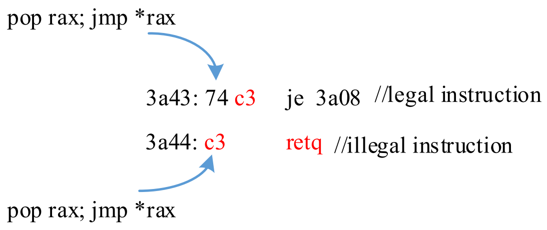


Fig. 19. Illegal jump generates an illegal instruction.

EPT page tables, and the two sets of EPT page tables require a total of 514MB of memory.

6.3. Comparison with existing methods

To evaluate existing security methods, we extend the evaluation method proposed in (Burow et al., 2017). We add a new evaluation indicator CFR (control flow tracking redundancy) on the basis of the evaluation indicators CF, RP, SAP.F, SAP.B and Q used in (Burow et al., 2017). A detailed description of all factors can be found in the Appendix A. The results of KPointer and existing methods are shown in Fig. 18. The results show that the overall performance of KPointer is relatively balanced.

KPointer detects whether suspicious control data is updated by monitoring the stack overwrite function. It can approach a scenario where an attacker exploits a stack overflow vulnerability to tamper with control data. Based on this method, KPointer can greatly reduce the redundancy of traced instructions, thereby improving the CFR score.

By migrating control flow into a separate space, KPointer can capture all ICT instructions (such as "*call* *" and "*jmp* *") including the illegal instructions shown in Fig. 19. Because in a separate space, the current address space is not the same as the original one. The operands of ICT instructions are stored in writable memory, and they still point to the original address space. So, the control flow will jump from the current address space to the original address space. Since the last page table PTE of the original code segment is unreadable, the control flow transfer will be captured. Therefore, the CFT performance of KPointer is good. Whether it is ROP, JOP, COOP, SROP or LOP, as long as the attacker uses the overflow vulnerability to tamper with the control data on the stack, KPointer can find it through control flow monitoring and tracing, and use the security detection strategies to determine its legitimacy. Therefore, KPointer has a good defense effect against stack overflow-based CRAs and their variants. In addition, KPointer does not rely on source code and can track the control flow transfer instructions in shared libraries. Based on these performances, KPointer improved its CFT, Q, AP.F and AP.B.

The reason why KPointer's AP.F performs well is that it only traces the ICT instructions after the stack overwrite operation, and uses a more granular security policy. Moreover, KPointer will detect whether the return address has been tampered with after the stack overwrite operation, which makes its AP.B perform good. Finally, according to the SpecCPU2006 test, KPointer only introduces less than 3% performance overhead to the CPU. Therefore, its RP performance is also good.

On the whole, KPointer is better than the current comprehensive methods (such as IFCC, O-CFI and PathArmor) when fighting against the CRAs based on stack overflow. To the best of our knowledge, most security methods (such as πCFI, MCFI, HW-asst, and CCFI, etc.) fail to detect the illegal instructions converted from operands (shown in Fig. 19). Because they only set checkpoints at the locations of legal instructions (opcodes), and ignore the operands with specific forms (ie, illegal instructions, such as *c3*, whose binary forms are the same as the jump instructions, but they are only operands rather than opcodes.). If an active method (such as a compiler-based method) attempts to detect all illegal instructions, the checkpoints should be set at all the positions of the operands with specific forms. Due to the unaligned nature of x86 code, the method requires a byte-by-byte search for all possible illegal instructions. There are a large number of such operands, which can seriously affect the execution efficiency of applications. Missing illegal instructions not only lowers CFT, but also lowers Q, AP.F, and AP.B.

Moreover, the compiler-based methods, such as IFCC, πCFI, MCFI, CCFI, and VTI, require the support of source code, which leads to the protection failure on the loaded libraries. Kernel-CFI needs to analyze the pointers in the source code of kernel, which is invalid for other kernel objects (such as loadable kernel modules) that have no source code. These characteristics can negatively impact on their CFT, Q, AP.F and AP.B.

Both πCFI and MCFI are context-sensitive methods, and their Q and AF.B have good performance. In general, the shadow stack can retain the original return address, which is helpful for improving AF.B. Lockdown has a higher AF.B based on this method. The methods using hardware-assisted techniques can achieve better results with less overhead, such as O-CFI using MPX and PathArmor using LBR. Comprehensive and reasonable security policies have a positive effect on improving the Q, AP.F and AP.B. For example, HW-asst can ensure the correctness of the return address by verifying each instruction *ret*, and it has a better AP.B. In contrast, O-CFI employing address randomization can be bypassed by code probing, which affects its Q. At the same time, O-CFI builds a target set for jump instructions, but the elements in the set are not unique. Therefore, its AF.B and AF.F do not perform well. Since CFIGuard detects all indirect jumps, it has better Q. However, this method relies on high-precision CFG. Especially for shared libraries that do not contain source code, the jump relationship between code blocks is not clear, which may be changed with the input and conditions. A low-precision CFG will reduce the AP.B and AP.F of CFIGuard. If a method can approximate the real attack scenario to the greatest extent and detect the code in the scenario in a targeted manner, it can get an ideal CFR. On the contrary, the more attack-agnostic code involved in security methods, such as HW-asst and CFIGuard, the lower their CFR.

Although KPointer has some advantages when defending CRAs based on stack overflow, it still has some shortcomings. First and foremost, KPointer is only defensive against stack overflow based CRAs. If the tampered control data is on the heap, the KPointer has no effect. Second, in special execution scenarios, KPointer will misjudge. If a function uses a full copy method (such as *memcpy(struct_pointer, memory_pointer, sizeof(struct xxxx))*) to update the data structure (containing a function pointer that has been assigned), when the function pointer in the data structure is used as an operand, KPointer will judge the current legal operation as illegal. Third, in special execution scenarios, KPointer will miss judgment. When an attacker can construct all gadgets "*jmp* *" inside a same function, and he will not destroy the pointing feature of control data and the jump feature of ICT instructions, KPointer cannot judge they are illegal. If the function pointer array in the function stack frame is stored adjacent to a piece of control data, the control data (keeping the same pointing characteristics after being updated) cannot be judged to be illegal even it has been tampered with.

These weaknesses can negatively impact CFT, Q and AP.F. If the corrupted control data is on heap, KPointer cannot detect the control data's update, and it will not trace the subsequent ICT instructions. Therefore, CFT, Q and AP.F are affected. Misjudgment directly affects AP.F, while omission directly affects Q. In fact, misjudgments and omissions are rare. We ran Nginx, Redis and SpecCPU for more than 12 hours respectively, and KPointer did not have any misjudgments or omissions.

6.4. Limitations

Currently, KPointer still has some inherent flaws. First, KPointer is only defensive against stack overflow-based CRAs. If the tampered control data is on the heap, KPointer has no effect. Second, it only has a protection effect on user space code. Compared with user space, the function calls in kernel space do not require the participation of PLT and GOT, which makes our security strategies invalid. In fact, KPointer is also effective for backward control flow in the kernel. For forward control flow, KPointer needs to adopt new tracking methods and security strategies. Third, KPointer only supports Linux under the x86 architecture with VMX and EPT, which is invalid for the ARM architecture and Windows. We plan to expand KPointer to ARM architecture and Windows in our future work.

Moreover, as described in Section 6.3, KPointer has the possibility of misjudgment and omission in special execution scenarios. Fortunately, we have not found the special execution scenarios in real applications (such as *SpecCPU2006, Lmbench, Nbench, Memmeter, Ngnix, Apache*, and *Redis*). Because, after the data structure is assigned and before being reclaimed, it rarely updates its member variables in a complete copy manner. Typically, overwriting data structures with *memcpy* only happens during initialization. At this moment, the function pointer in the data structure has not been assigned an address (usually its value is 0). Therefore, it does not become the object to be detected, and there is no false positive. When a specific member variable in the data structure needs to be updated, the user generally assigns the target value directly to it, rather than completely overwriting the entire data structure. Furthermore, the attacker constructs all gadgets in a single function, and these gadgets conform to the specific code form and behavioral capabilities required for the attack, which is almost impossible. The reason is that the gadgets that match the attack form and attack capability have strict screening conditions. Therefore, the attacker needs a large amount of binary code to filter out enough qualified gadgets. The amount of binary code contained in a single function is difficult to meet this requirement. In short, KPointer has the possibility of misjudgment and omission in theory, but it rarely occurs in real execution scenarios.

7. Conclusion

This paper proposes a security method KPointer to defend against CRAs based on stack overflow. KPointer finds suspicious data by detecting the functions that can overwrite stack data. After that, a new code space was established to track the transfer path of the suspicious data and locate the ICT instructions related to the suspicious data. Finally, KPointer uses the correlation between instructions and data to build security strategies to detect the legitimacy of ICT instructions. The experiment and analysis show that KPointer has a good defense effect on CRAs based on stack overflow. It introduces 2.7% performance overhead to the CPU.

Declaration of Competing Interest

We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work. And there is no professional or other personal interest of any nature or kind in any product, service and/or company that could be construed as influencing the position presented in, or the review of, the manuscript entitled, “KPointer: Keep the code pointers on the stack point to the right code”.

CRediT authorship contribution statement

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

Appendix A

In fact, the six indicators (CFT, CFR, RP, Q, AP.F, AP.B) are all inspired by Burow[44]. Except for the RP, other evaluation indicators are obtained according to the qualitative analysis of the security methods' protypes. Their specific meanings are as follows:

Table 5
The functions detected by KPointer.

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

CFT: Control Flow Transfer. It refers to jump instructions that can be tracked by security tools, which may be used in CRAs. The traced instructions include *"call *%register", "call *(%register)", call *value(%register), "call *(%register, %register, value)", "call *pointer", " jmp *%register", "jmp *(%register)", "jmp *address(, %register, value)", "ret", "retn value"*, and *"retf value"*. The more indirect jump instructions a security tool can trace, the higher the CFT score. In fact, these instructions may be legal instructions or illegal instructions generated by an attacker through an arbitrary jump. An example is shown in Fig. 19. In a normal execution scenario, the control flow would jump to the address *3a43* via the instruction "*jmp *rax*". Then, the code "*74 c3*" ("*je 3a08*") will be executed. If the attacker tampered with the control data to make the control flow jump to the address *3a44*, the code to be executed becomes "*c3*" ("*retq*"). Actually the "*c3*" in the legal code "*74 c3*" is only an operand (a relative offset to the current location), not an opcode. Such an instruction is called illegal instruction. This reduces CFT if the security scheme cannot track and detect such illegal instructions. Additionally, the instructions being traced may be located in shared libraries that have no source code and have been loaded into memory. A method that fails to track instructions in a library lowers the CFT score.

CFR: Control Flow Redundancy. It refers to the additional control flow introduced by security methods to track and detect the legitimacy of potential attack targets, whose jump instructions can be hijacked by attackers. For example, to check the instruction "*ret*", the security method rewrites the instruction with some code to check its return address; if the return address used by "*ret*" is impossible to be tampered with, the checking code will generate redundant control flow. In general, security methods assume that every control flow transfer instruction can be used by attackers potentially. Therefore, all control flow transfer instructions are traced or checked. However, in a real attack scenario, only those code blocks whose control data or condition data can be tampered with can become attack vectors. So, many checks for control flow are meaningless. The main factor affecting CFR is the accuracy of identifying attack scenarios. The more we can approximate real attack scenarios and accurately detect potential attack targets, the more we can reduce control flow redundancy and thus improve CFR.

RP: Reported Performance. It refers to the performance overhead reported in the paper. The lower the performance overhead, the higher the score.

Q: Qualitative Security. Generally speaking, the factors affecting Q mainly include three aspects: the attack principles, the defense principles and the characteristics of the objects to be attacked. Therefore, we qualitatively analyze Q around the three aspects. If we can identify all possible attack scenarios and make the jump instructions in attack scenarios only jump to legitimate targets, we can get the ideal Q. So, we define Q as follows:

$$Q = \sum_{n=1}^{tar\_num} P_n * \frac{1}{L_n}$$

Tar_num represents the total number of control flow transfer instructions. *Pn* represents the probability of the n^{th} control flow transfer instruction that can be maliciously used. It is determined by the attack principles and the characteristics of the objects to be attacked. *Ln* represents the jump target number of the n^{th} control flow transfer instruction in the attack scenario. It is determined by the defense principles of the method to be evaluated. For the code that cannot be used by an attacker, its *Pn* is 0. Therefore, it does not have any effect on improving Q, which is reasonable to portray Q. The larger the *Pn*, the greater the risk of the n^{th} instruction. So, the more a security method protects this instruction, the more its effort contributes to improving Q. In practice, each control flow transfer instruction has only one jump target when it is executed. In other words, the ideal value of *Ln* is 1. The larger the *Ln*, the smaller the Q, which is in line with the security meaning of Q.

AP.F: Analysis Precision for Forward Control Flow. In fact, not all forward control flow can be constructed as gadgets. A forward jump instruction used as a gadget must satisfy: 1) its operand (ie, the jump target) must be located in writable memory; 2) the operand can be tampered with. These two conditions must be met at the same time, otherwise it cannot be used as a gadget. For example, since the jump target "*address*" in the instruction "*call address*" is fixed to a non-writable code segment, this instruction cannot be used as a gadget. The closer the target to be traced by the security method is to these two conditions, the higher its AP.F score.

Furthermore, in modern program structures, there are some implicit jump rules for forward control flow, such as the rules in Section 5.4. For security methods that focus on jumping rules, the closer they can approximate these rules in judging the legality of control flow, the higher the AP.F score.

Moreover, it is legal for a forward jump instruction to have one and only one jump target each time it is executed. For security methods that set an instruction boundary, the closer they approximated this ideal instruction boundary, the higher the AP.F score.

AP.B: Analysis Precision for Backward Control Flow. Similar to AP.B, not all backward jump instructions can be used as gadgets. Only if the return address has the possibility of being tampered with, the return instruction corresponding to the return address can be used as a gadget. The more a security method can approximate the execution scenario of tampering with return address, the higher its AP.B score.

In addition, whether the security method can accurately judge the legitimacy of the return address before the control flow returns is also an important basis for evaluating AP.B. For example, canary-based methods can cause false positives due to the leaked canary value, which results in a lower AP.B score.

Appendix B

The functions in Table 5 can write data to the stack, and they are all detection objects of KPointer. Under certain conditions, they may cause stack overflows. Although some functions (such as *getc()*) cannot directly cause stack overflow, they are possible in a loop code block. For example, in a looping code block, the number of *getchar()* calls controlled by the input, and it can write a non-fixed-size string to the stack, which may also cause a stack overflow.

References

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

Bittau, A., Belay, A., et al., 2016. Hacking blind. In: Proc. *the IEEE Symposium on Security and Privacy*, pp. 227–242.

Bittau, A., Belay, A., 2016. Hacking blind. In: Proc. *the IEEE Symposium on Security and Privacy*. IEEE, pp. 227–242.

Bounov, Dimitar, et al., 2016. Protect-ing C++ dynamic dispatch through VTable interleaving. In: Proc. NDSS.

Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M., 2017. Control-flow integrity: Precision, security, and performance. ACM Comput. Surv. 50 (1), 1–33.

Cowan, C., Pu, C., Maier, D., et al., 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow at-tacks. In: Proc. *USENIX Security Symposium*, 98, pp. 63–78.

Criswell, J., Dautenhahn, N., 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In: Proc. *IEEE Symposium on Security and Privacy*, pp. 292–307.

Curtsinger, C., Berger, E.D., 2013. STABILIZER: Statistically sound performance evaluation. Proc. Acm Sigarch Comput. Architect. News 41 (1), 219–228.

Fan, X., Sui, Y., 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In: Proc. t*he 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 329–340.

Fu, J., Jin, R., Lin, Y., et al., 2018. Function risk assessment under memory leakage. In: Proc. *International Conference on Networking and Network Applications (NaNA)*, pp. 284–291.

Fu, J., Jin, R., Lin, Y., et al., 2018. Function risk assessment under memory leakage. In: Proc. *International Conference on Networking and Network Applications (NaNA)*, pp. 284–291.

Fu, J., Lin, Y., Zhang, X., 2016. Code reuse attack mitigation based on function randomization without symbol table. In: Proc. *IEEE Trustcom/BigDataSE/ISPA*, pp. 394–401.

Gawlik, R., Kollenda, B., Koppe, P., Garmany, B., Holz, T., 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. Proc. NDSS 16, 21–24.

Gawlik, R., Kollenda, B., Koppe, P., et al., 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In: Proc. *the NDSS*, pp. 21–24.

Ge, X., Talele, N., Payer, M., et al., 2016. Fine-grained control-flow integrity for kernel software. In: Proc. *IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 179–194.

Guo, Y., Chen, L., Shi, G., 2018. Function-oriented programming: A new class of code reuse attack in c applications. In: Proc. *IEEE Conference on Communications and Network Security*, pp. 1–9.

Guo, Y., Chen, L., Shi, G., 2018. Function-oriented programming: A new class of code reuse attack in c applications. In: Proc. *IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9.

Gupta, A., Kerr, S., Kirkpatrick, M.S., et al., 2013. Marlin: A fine grained randomization approach to defend against ROP at-tacks. In: Proc. *International Conference on Network and System Security*, pp. 293–306.

Hiser, J., Nguyen-Tuong, A., Co, M., et al., 2012. ILR: Where'd my gadgets go? In: Proc. *Symposium on Security and Privacy*, pp. 571–585.

Hu, H., Qian, C., Yagemann, C., et al., 2018. Enforcing unique code target property for control-flow integrity. In: Proc. *the ACM SIGSAC Conference on Computer and Communications Security*, pp. 1470–1486.

Kangjie, Lu†, S, Nürnberger, Backes, M., et al., 2016. How to make ASLR win the clone wars: Runtime re-randomization. In: Proc. *23rd Network and Distributed System Security Symposium (NDSS)*.

Khandaker, Mustakimur Rahman, Liu, Wenqing, et al., 2019. Origin-sensitive control flow integrity. In: Proc. *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 195–211.

Khandaker, M., Naser, A., Liu, W., et al., 2019. Adaptive call-site sensitive control flow integrity. In: Proc. *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 95–110.

Kwon, D., Shin, J., Kim, G., et al., 2019. {uXOM}: Efficient {eXecute-Only} Memory on {ARM}{Cortex-M}. In: Proc. *the 28th USENIX Security Symposium (USENIX Security 19)*, pp. 231–247.

Lan, B., Li, Y., Sun, H., et al., 2015. Loop-oriented programming: A new code reuse attack to bypass modern defenses. In: Proc. *IEEE Trustcom/BigDataSE/ISPA*, 1, pp. 190–197.

Larsen, P., Franz, M., 2020. Adoption challenges of code randomization. In: Pro. *the 7th ACM Workshop on Moving Target Defense*, pp. 45–49.

Liljestrand, H., Nyman, T., Wang, K., Perez, C.C., Ekberg, J.-E., Asokan, N., 2019. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In: Proc. *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 177–194.

Li, Y., Wang, M., et al., 2020. Finding cracks in shields: On the security of control flow integrity mechanisms. In: Proc. *The ACM SIGSAC Conference on Computer and Communications Security*, pp. 1821–1835.

Li, Y., Dai, Z., Li, J., 2018. A control flow integrity checking technique based on hardware support. In: Proc. *3rd Advanced Information Technology, Electronic and Automation Control Conference*, pp. 2617–2621.

Lu, K., Song, C., et al., 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In: Proc. *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 280–291.

LucasDavi, Ahmad-RezaSadeghi, Daniel, Lehmann, Fabian-Monrose, 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: Proc. *the 23rd USENIX Conference on Security Symposium*, pp. 401–416.

Marco-Gisbert, H., Ripoll, Ripoll I., 2019. Address space layout randomization next generation. Applied Sciences 9 (14), 2928–2952.

Mashtizadeh, A.J., Bittau, A., Boneh, D., et al., 2015. CCFI: Cryptographically enforced control flow integrity. In: Proc. *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 941–951.

Mohan, V., Larsen, P., Brunthaler, S., et al., 2015. Opaque control-flow integrity. In: Proc. *NDSS*, 26, pp. 27–30.

Niu, B., Tan, G., 2014. Modular control-flow integrity. In: Proc. *the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 577–587.

Niu, B., Tan, G., 2015. Per-Input Control-Flow Integrity. In: Proc. *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 914–926.

Oikonomopoulos, A., Athanasopoulos, E., Bos, H., et al., 2016. Poking holes in information hiding. In: Proc. *the 25th {USENIX} Security Symposium*, pp. 121–138.

Payer, M., Barresi, A., et al., 2015. Fine-grained control-flow integrity through binary hardening. In: Proc. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 144–164.

Qiu, P., Lyu, Y., Zhai, D., et al., 2016. Physical unclonable functions-based linear encryption against code reuse attacks. In: proc. *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6.

Riq, Gera Riq "Advances in format String Exploitation.", Ruben Boonen. https://www.fuzzysecurity.com/index.html Accessed( ).

Sui, Y., Ye, D., Su, Y., Xue, J., 2016. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. IEEE Trans. Reliabil. 65 (4), 1682–1699.

Szekeres, Laszlo, Payer, Mathias, et al., 2013. SoK: Eternal war in memory. In: Proc. *IEEE Symposium on Security and Privacy*, pp. 48–62.

Tice, C., Roeder, T., et al., 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In: Proc *the 23rd USENIX Conference on Security Symposium*, pp. 941–955.

Van der Veen, V., Andriesse, D., et al., 2015. Practical context-sensitive CFI. In: Proc. *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 927–940.

Wilander, J., Nikiforakis, N., Younan, Y., et al., 2011. RIPE: Runtime intrusion prevention evaluator. In: Proc. *the 27th Annual Computer Security Applications Conference*, pp. 41–50.

Ye, D., Su, Y., et al., 2014. WPBOUND: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In: Proc. *the International Symposium on Software Reliability Engineering*, pp. 88–99.

Yuan, P., Zeng, Q., Ding, X., 2015. Hardware-assisted fine-grained code-reuse attack detection. In: Proc. *International Symposium on Recent Advances in Intrusion Detection*, pp. 66–85.

Zhang, M., Sekar, R., 2013. Control flow integrity for {COTS} binaries. In: Proc. *22nd {USENIX} Security Symposium*, pp. 337–352.

Zhang, Chao, Wei, Tao, Chen, Zhaofeng, et al., 2017. Practical control flow integrity and randomization for binary executables. In: Proc. *IEEE Symposium on Security and Privacy*, pp. 559–573.

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

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

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

Yi Lu, graduated from the Chengdu University of Information Technology in 2020. Now, he is a graduate student at the School of Computer Science and Technology in the China University of Mining and Technology. His research interests include code optimization, and cloud computing.

ShanQing Guo is currently a professor with the School of Cyber Science and Technology at Shandong University. He received his M.S. and Ph.D. degrees in computer science from Ocean University, China, in 2003, and Nanjing University, China, in 2006 respectively. He joined the School of Computer Science and Technology at Shandong University as an assistant professor in 2006. His research interests include AI Security, Data-driven Security, Software and System Security. He has published in TSE, TDSC, S&P, USENIX Security, ICDE and other venues. He also serves as a program committee member or a reviewer for various international conferences and journals, e.g., ISSRE, ICSME and Computer & Security.

GuoYuan Lin, received the PhD degree from the Nanjing University in 2011. Now he is the deputy dean of School of computer science and technology of China University of mining and technology. He has long been engaged in the research of cyberspace security, information security, cloud computing, cloud security, information project research and development, operating system architecture and system security.