Stack overflow vulnerabilities are among the most common security issues. However, the existing stack overflow detection solutions only protect the return address and ignore the imbalance between function calls and returns in the system, which will lead to a higher false-positive rate. In this paper, we propose an instruction reorganization virtual platform technique for kernel stack overflow detection, named IRePf. It can dynamically monitor the kernel stack when the system is running through dynamic reorganization instructions, thread creation and termination, call instructions, and RET instructions. IRePf uses backup stack creation and destruction technology to back up the return address and the address of the return address at the same time. IRePf determines whether the stack overflow occurs in the system when the function call and return are unbalanced to realize the kernel stack overflow detection. The experimental results show that IRePf can effectively detect stack overflow attacks, has low system resource occupancy and high real-time performance, and effectively improves the ability and security of defense stack attacks.

1. Introduction

The computer has become an information processing tool widely used in various fields. However, with the development of the Internet and the enhanced impact of the network on society, computer viruses, malicious codes, and network vulnerabilities have been threatening the security of computers [1]. Buffer overflow is a common method of exploiting system vulnerabilities to attack computers [2]. By writing data exceeding the defined length into the buffer, overwriting the data of adjacent storage units, and destroying the stack of the program, a program is forced to execute the processes of an attacker. The defined shell code is used to implement the attack.

Buffer overflow attacks are usually divided into stack overflow attacks and heap overflow attacks. Since stack overflow attacks account for a larger proportion than heap overflow attacks and the methods of stack overflow attacks are more diverse and more subtle, this paper mainly studies the detection system methods relevant to stack overflow attacks. In 2005, stack overflow-related vulnerabilities accounted for approximately 40 percent of SANS’ selection of the 20 most threatening vulnerabilities. Google Chrome was also exposed in 2012 to a stack overflow vulnerability that caused the browser to deny service. According to statistics, from 2006 to 2017, although the number of stack overflow attacks did not increase significantly, the proportion of stack overflow attacks becoming high-risk vulnerabilities is increasing. Stack overflow attacks exploit a kind of computer system vulnerability that is extremely difficult to detect and defend against. At present, system attacks using stack overflow account for more than 50% of all attacks. Stack overflow attacks may cause problems such as program failure, system crash, and network paralysis. In more serious cases, attackers can obtain hosts or servers through stack overflow attacks, along with partial or total control. In addition, stack overflow attacks widely exist in various operating systems and applications, and there are various attack methods. However, most of the existing stack overflow detection and defense technologies only protect the
return address in the system. The problem of unbalanced function calls and returns in the system is indeed ignored, and it is impossible to judge whether stack overflow occurs. Therefore, there are still many difficulties in fundamentally solving the problems of stack overflow detection and defense.

Based on the in-depth study of the existing stack overflow attack detection technology, this paper proposes an instruction reorganization virtual platform, IRePf, for kernel stack overflow detection. IRePf can dynamically monitor the kernel stack while the system is running by dynamically reorganizing instructions, thread creation and termination, call instructions, and RET instructions. IRePf then uses backup stack establishment and destruction technology to back up the return address and the address of the return address at the same time and judges whether system stack overflow occurs when the function call and return are unbalanced in order to realize kernel stack overflow detection. The main work of this paper is summarized as follows:

1. The proposed IRePf can monitor machine instructions in real time while the computer is running, providing an effective method for monitoring the kernel stack.

2. The monitoring methods in IRePf are introduced. These methods can monitor the creation and termination of threads, call instructions and RET instructions to monitor the initialization, and destruction and data modification of the stack in real time.

3. A method of creating and destroying a backup stack is proposed. By backing up the return address and the address of the return address at the same time, the problems of unbalanced function calls and returns in the system are solved.

However, there are still many efficiency optimization problems to be solved in IRePf, especially the optimization of the instruction reorganization virtual platform method and further optimization of the search algorithm. There is still much room for improvement in the practicality of the technology.

The remainder of the paper is organized as follows: Section 2 outlines the main research content of related work. Section 3 briefly introduces the method structure based on IRePf. Sections 4 and 5 explain the IRePf framework and IRePf-based stack overflow detection techniques in detail. Section 6 verifies the validity of the experimental results. Section 7 concludes this paper and proposes future work.

2. Related Work

Stack overflow has caused a series of security problems and attracted widespread attention from all parties. In recent decades, researchers have conducted many in-depth studies on stack overflow and proposed a large number of stack overflow detection and defense techniques. One [3] first analyzed the structure of the stack under the Linux system and how to use the stack buffer overflow to attack. Jack [4] proposed the idea of jumping to malicious code using the instruction “jmp esp” in the system DLL on the Win32 platform. At present, stack overflow detection technology is divided into stack static detection and stack dynamic detection.

2.1. Stack Static Detection Technology. The detection of stack buffer overflow by static detection technology does not depend on the operation of the program, and static detection can usually be performed on the program before and after compilation. Stack overflow detection tools IRePf [5] and ARMORY [6] can detect stack overflow vulnerabilities by inputting source code, building a vulnerability database, scanning source code, and matching vulnerability patterns; Meng et al. [7] implemented semi-supervised learning and AST analysis extracted from predicted buffer overflow vulnerabilities; Padmanabhan and Tan [8] detected buffer overflow by mining static code properties, extracted static by identifying potentially vulnerable sentence structures in binary program code attributes, and used data mining methods to analyze their attributes; Dudina and Belaventsev [9] proposed an interprocedural path and context-sensitive overflow detection algorithm, which is a path-sensitive static analysis based on symbolic execution and state merging, which can find incorrectly defined program points in C/C++ source code; Meng et al. [10] proposed a machine-learning-based static analysis method to assist in auditing buffer overflow vulnerabilities. Seven static attributes are extracted, and these attributes are embedded into a vector space to train machine learning algorithms.

2.2. Stack Dynamic Detection Technology. The detection of stack buffer overflow by dynamic detection technology depends on the running of the program. It usually needs to insert some instructions into the program or modify some instructions to detect stack overflow. Solanki et al. [11] focused on stack-based buffer overflow attacks and protecting global offset tables; they used a shadow stack (a kind of linked list that behaves as a stack), stored return addresses and frame pointers to prevent buffer overflow attacks, and compared these return addresses and frame pointers at the beginning and end of the program. Cowan et.al. proposed a stack overflow detection tool, PointGuard [12], which inserts the word “canary” in the function pointer to protect the function pointer from being overwritten by stack overflow. Rao et al. [13] proposed a protection technique called BFWindow for performance- and resource-sensitive embedded systems, which speculatively checks the protection of data attributes by coloring the data structure and expanding the target memory block. For consistency in verifying each memory write, Sidiroglou et al. [14] proposed a method to replace the stack with a heap, namely, DYBOC technology; Prasad and Chiueh [15] constructed a binary code disassembly and instrumentation based tool, RAD (Return Address Defense), which does not rely on source code for stack overflow detection; Mouzarani et al. [16] proposed a new intelligent fuzzing method for detecting stack-based buffer...
overflows in binary code, which uses concrete symbols to compute the path and vulnerability constraints of each execution in the program and then exploits the vulnerability constraints to detect buffer overflows in feasible execution paths of programs; Love et al. [17] proposed a new method for buffer overflow detection, which extracts the static variable space in the stack by tracking the frame pointer operation in the instruction, stores this information in the variable record table (VRT), and inserts parameters in the VRT to check for buffer overflows; Fen et al. [18] proposed a randomization method based on data protection, which uses a 32-bit random number mask and XOR to protect arrays and pointers; Gao et al. [19] analyzed the accessibility of overflow vulnerability warnings and executed symbolic source code to verify and classify warnings by checking that the warnings satisfy all path conditions and buffer overflow constraints.

Some researchers combined the advantages of static and dynamic detection analyses to detect buffer overflow vulnerabilities. Jang and Choi [20] proposed a method to detect and prevent buffer overflow vulnerabilities in C/C++ programs, which generates replacement code, compares the result of each replacement variable with the expected result of the input data, and executes detection failures before statements and prevents malicious access. Jiadong et al. [21] proposed a buffer overflow vulnerability prediction method, which statically extracts software metrics from software source code and dynamically extracts functional-level data from dynamic data streams and builds a model based on a decision tree algorithm, which measures buffer overflow vulnerabilities at the functional level. Padmanabhu and Tan [22] proposed a hybrid approach combining static and dynamic analysis with machine learning to audit buffer overflow vulnerabilities, which uses simple rules and dynamic analysis to identify overflow vulnerabilities and mines static code properties to detect buffer overflow vulnerabilities. Regarding predictive dynamic analysis of statements that do not identify vulnerabilities, Tian et al. [23] proposed a dynamic buffer overflow prevention system utilizing virtualization technology, which utilizes static binary analysis to identify instructions offline and combines virtual machines and online patching to dynamically detect protected running programs, making the system easy to deploy without restarting the program.

In addition, researchers have also explored the feasibility of hardware optimization. Chen et al. [24] used virtualized memory access techniques to automatically diagnose and patch stack overflow vulnerabilities by moving vulnerable buffers to protected memory regions; Danger et al. [25] modified the processor architecture to mask the immediate return address of the program, which prevents buffer overflows from overwriting the return address of the subroutine parent. Nashimoto et al. [26] proposed a method with multiple fault injection. The buffer overflow attack is an effective combination of hardware and software attacks and return address attacks, and the branch judgment operation and loop counter update countermeasures against the above attacks are proposed.

In summary, although stack overflow detection has achieved certain achievements in static, dynamic, and combined technologies, there are still limitations and room for improvement in whether the detection is based on the application of source code or binary code technology. At present, the existing static detection technology cannot detect stack overflow when the program is running, and the dynamic detection remains primarily theoretical and cannot be applied in practice. At the same time, the existing stack overflow detection only protects the return address, ignoring the general situation. The existing function call and return are not harmonious, so the stack overflow detection effect is not significant. With the emergence of machine learning, stack overflow detection technology that monitors runtime instructions from a dynamic perspective will provide the possibility for traditional stack overflow detection technology to achieve higher practicability and effectiveness.

3. Method Architecture

3.1. Method Construction. In the operating system, the initialization of the stack will create two stacks: one for the kernel service, called the kernel stack, and one for the upper-layer application service, called the user stack. Because kernel stack overflow is more difficult to detect than user stack overflow and causes more harm, this paper mainly studies the detection technology of kernel stack overflow. Unless otherwise specified, stack overflows mentioned in subsequent articles refer to kernel stack overflows. The user stack overflow detection is similar to the kernel stack overflow detection method and can be obtained in the same way.

To dynamically detect the stack overflow in the system, the IRePf proposed in this paper can monitor the operation of the instruction in real time when the computer is running and further monitor the change of the thread. First, it is necessary to dynamically monitor the changes in the kernel stack. During the operation of the system, the changes in the function call stack mainly include the creation (initialization) of the stack, the termination of the stack, and the change in the stack content. The creation and termination of the thread will lead to the creation and termination of the stack, and the function call in the thread will change the content of the stack. Therefore, the change in the kernel stack can be translated into the creation and termination of the thread and the function call in the thread. Second, stack overflow is detected to determine whether the stack data have changed. IRePf establishes a backup stack for each thread when it is created. When a function is called in a thread, IRePf backs up the stack data. When a function call returns in a thread, IRePf compares the backup stack data with the actual stack data to detect stack overflow. When a stack overflow occurs, IRePf uses the backup stack data to restore the damaged real stack data to ensure that the tampered data can be recovered, thus achieving the effect of stack overflow detection and defense. According to the above analysis, the architecture of the kernel stack overflow detection method based on IRePf is shown in Figure 1.
3.2. Function Call and Return Imbalance. When a function call in a thread occurs, there is an imbalance between the function call and the return. Usually, the numbers of the two types of instructions are equal, but, in some cases, the number of RET instructions may be greater than the number of call instructions, which is called function call and return imbalance. For example, in the macro implementation of the switch_to function in ULK, when process A calls the switch_to function to switch to process B, it may not use the call instruction to process the call operation but first pushes the EIP of process B and then jumps to the called function; finally, after the switch_to function call ends, the RET instruction is used to return, and the above function call has the RET instruction but not the call instruction, which yields the case where the numbers of call and RET instructions are not equal.

4. IRePf Framework

IRePf first obtains the next instruction fragment by monitoring the running instructions of the computer in real time. Then, the obtained instruction fragments are analyzed, jump instructions are found, and the original instructions are modified by inserting custom control instructions before the jump instructions. Finally, the system executes the modified instruction fragment to achieve the purpose of the control instruction.

IRePf monitors the running instructions of the computer, caches the running environment of the current instruction, and pushes the register data related to the running instruction to the buffer stack, including the values of the address register and other registers. Then, we obtain the address of the instruction to be scheduled from the address register and use the jump instruction as the retrieval target, treat the jump instruction and all previous machine instructions as one instruction fragment, and insert the push stack before the last instruction of the fragment instruction and the second jump instruction.

As shown in Figure 2, the next run instruction of fragment A is fragment B. IRePf reassembles A and B to generate reassembled instruction fragments A″ and B″. The push stack instruction saves the target address and parameters of the first jump instruction of A and B so that, after A″ runs, IRePf can obtain the entry address of B. The second jump instruction points to the entry address of IRePf so that, each time, each reorganized instruction fragment returns to IRePf after execution to continue analyzing the next instruction fragment to be scheduled.

Due to the high repetition of machine instructions during program running, IRePf adds an address mapping table for optimization before obtaining the instructions to be scheduled. The address mapping table is established and maintained by IRePf. It adopts a variable-length linked list structure, and the occupied space can be released. As shown in Figure 3, the format of the address mapping table is (first address, second address), which indicates whether the instruction segment corresponding to the first address has a saved reassembled instruction segment. If the corresponding record is found, the green line marking process is performed. Otherwise, IRePf will perform the purple line marking process.

Algorithm 1 shows the instruction reorganization process in IRePf. IRePf takes jump instructions (including call instructions and jump instructions) as the retrieval target, divides instruction fragments, and reorganizes instructions, which will greatly improve the efficiency of reorganization. Since IRePf avoids monitoring and reorganizing each instruction, its operation does not have much impact on system overhead.

Algorithm 2 shows the instruction modification process of IRePf, including inserting push stack instruction and second jump instruction. At the same time, the generated
reassembly instruction fragments are stored in a separate reassembly space, and the address mapping table is updated.

5. Stack Overflow Detection Technology Based on IRePf

5.1. Thread Initialization and Termination Monitoring Technology. This article monitors stack initialization and termination by monitoring thread creation and termination. Threads can be divided into system threads and user threads. The creation and termination of system threads are performed in the kernel space Ring0, and the creation and termination of user threads are performed in user space Ring3 and kernel space Ring0. Figures 4 and 5 show the creation processes for system threads and user threads. As seen from the figure, as long as the return value of PspCreateThread, KeInitThread, or Ke386InitThreadWithContext is true, a new thread will be generated and started soon. Therefore, the monitoring method of thread creation is to use IRePf to hook the return points of these three functions to complete the real-time monitoring of thread initialization.

Figures 6 and 7 show the termination process of system threads and user threads. As seen from the figure, a knowable thread will exit as soon as any function of PspTerminateThreadByPointer, PspExitThread, or KeTerminateThread is created. Therefore, the monitoring method of thread termination is to use IRePf to hook the call points of these three functions to complete the real-time monitoring of thread termination.

5.2. IRePf Stack Overflow Detection Technology. The function call in the thread will cause the data in the stack to change. To determine whether there is a stack overflow, it is necessary to monitor the call and return of the function. Therefore, we
use IRPeF to monitor the call and RET instructions to dynamically monitor the stack data changes, including stack initialization, termination, and content changes. The existing stack overflow attacks are all attacks that destroy the return address in the stack. The stack overflow detection and monitoring of whether the return address changes can be realized by backing up the return address. In addition, due to the problem of call and return imbalance in function calls, this paper adopts the method of backing up the address of the return address when the function is called to solve this problem. We use IRPeF to back up the return address and the address of the return address and then compare it with the

---

**Algorithm 1: Instruction reorganization.**

Input: source code $scode$
Output: reorganization code $rcode$

1. $inaddress$,
   $flag = true$, $i ← 0$, $j ← 0$/$inaddress$ is the entry address of $scode$.
2. $addrestable[n][2] ← (firstaddress, secondaddress)$
3. $re_fragment[n][2] ← (rcodep, secondaddress)$//reorganization instruction fragment $rcodep$
4. While $flag$ do
5.   If $scode$ all run then
6.     $flag = false$
7.   else
8.     Buffering the running environment of instructions
9.     $firstaddress ← inaddress$ or $firstaddress ← the address and parameters of jump instructions in stack$
10. If MATCH ($addrestable[i][1], firstaddress$) then
11. $secondaddress ← addressable[i][2]$
12. Restoring the running environment and jumping to the secondaddress to continue execution.
13. else
14. Disassembling the machine instructions to be executed from until the obtained instruction is jump instruction ins
15. $rcodep ← The instructions from firstaddress to ins$
   Call Algorithm 2
16. Restoring the running environment and running $rcodep$
17. End If
18. End If
19. End While

**Algorithm 2: Instruction modification.**

Input: source code fragments $scodep$ and $ins$
Output: reorganization code fragments $rcodep$

1. $thirdaddress ← COMPUTER$ (Original address and parameters of $ins$)
2. If $thirdaddress$ is an immediate data then
3. $firstaddress ← thirdaddress$, call step (15) in Algorithm 1
4. else
5.   $firstins ← ins$//first jump instructions
6.   $newins ← PUSH thirdaddress$
7. $newscoderp ← Adding an instruction newins after the last instruction of scoderp$
8. $plataddress ← The entry address of IRPf$
9. $inins ← Adding conditional transfer instruction secondins, the jump address is plataddress$
10. $rcodep ← Adding an instruction inins after newins in newscodep$
11. $re_fragment[j][1] ← rcodep; re_fragment[j][2] ← secondaddress; j ← j + 1$
12. $addrestable[n][2] ← (firstaddress, secondaddress); false ← true$
13. End If

---

Figure 4: The creation process of system threads.
real stack. The addresses are compared to determine whether a stack overflow has occurred, the method of which is discussed in detail as follows.

When executing the call instruction, the system pushes the return address to the stack of the calling function, then executes the instruction "push %ebp," pushes the current extended base pointer register (EBP) value of the calling function into the stack, and then executes the instructions "mov %esp and %ebp." The purpose is to make the frame pointer ebp point to the esp of the calling function. These two instructions initialize the ebp of the calling function, indicating and repairing the bottom of the stack; finally, the stack pointer esp moves down to open the stack space of the called function, esp points to the top of the stack, and the address space between ebp and esp becomes the stack space of the called function. When the function returns, the stack release mainly includes the three following processes:

1. The system releases the current stack space. esp moves up until it reaches ebp at the bottom of its stack (i.e., ebp where the original calling function is saved).
2. Return to the original scene. First, the instructions "mov %ebp and %esp" are executed, assigning ebp to esp: that is, esp points to the bottom of the stack. Then, "pop %ebp" pushes the original ebp and restores the stack base address. At this time, ebp automatically adds 4, which becomes where the old ebp position was saved before a position above.
3. The RET instruction pops up the return address. When the called function ends, the instruction at the address becomes the next instruction to continue execution.

When the RET instruction is executed, esp is the address of the return address, and *esp is the return address. Typically, in the case of a call paired with RET, the value of esp is equal to the address of the backed-up return address, and if the two values are different, the RET instruction is redundant, and there is no corresponding call instruction. For the redundant RET instruction in the system, as long as it is not greater than the address of the backup return address, it is considered that no overflow occurs because the stack grows from a high address to a low address. For redundant RET instructions, if the value of esp is greater than the address of the backup return address, it proves that it has jumped to the position of other stack frames in the call stack and destroys the original call structure of the stack, thus indicating that there is a stack overflow in the system. If the value of esp is less than the address of the backup return address, it proves that the RET instruction is still in the current stack frame, and the call structure of the stack is not destroyed, thus indicating that there is no stack overflow in the system. This method solves the problem of how to judge whether stack overflow occurs in redundant RET instructions: that is, it solves the problems of unbalanced function calls and returns.

Figure 8 shows the principle of IRePf’s stack overflow detection technique. IRePf intercepts the call and RET instructions when the system is running and inserts custom control instructions before the call and RET instructions. The control instruction inserted before the call instruction pushes the return address and the address of the return address to the backup corresponding to the thread to which the call instruction stack belongs and updates the thread ID index table. The control instruction inserted before the RET instruction will pop out a return address and the address of the return address in the backup stack corresponding to the thread to which the RET instruction belongs, and IRePf compares them with the return address and return address of the current real stack to detect stack overflow.

Algorithm 3 is a comparison algorithm between the backup stack and the real stack, which determines whether a stack overflow attack occurs by comparing the difference between the backup stack data and the real stack data.

5.3. Backup Stack Creation and Destruction Technology. The establishment of the backup stack is the key to the entire stack overflow detection system method. The initialization of
the stack in the system will create two call stacks, a kernel stack with a size of KERNEL_STACK_SIZE for the kernel service and a user stack with a size of USERL_STACK_SIZE for the upper application service. Therefore, the creation of the backup stack can be understood as the process of creating a new memory space for the existing operating system kernel functions. This article first detects and protects the kernel stack, so the allocated memory space size is KERNEL_STACK_SIZE * 2. The reason why double space is needed is to manage the imbalance between function calls and returns when the thread summary function call occurs, and, at the same time, the call instruction does not match the redundant RET instruction. In this paper, the technology of backing up the address and the return address of the return address in the real stack of the thread ID to the backup stack of the corresponding thread ID, finding the corresponding backup stack by looking up the thread ID index table. When judging whether a stack overflow occurs, IRePf will obtain the relevant information of the RET instruction, including the thread ID of the RET instruction, pop the return address and the address of the return address from the backup stack, find the backup stack by looking up the thread ID index table, and return it back to the IRePf to perform stack overflow detection to improve detection efficiency. The key code for

![Figure 8: The principle of IRePf's stack overflow detection technique.](image-url)
modifying the backup stack through call and RET instructions is shown in Figure 11.

If a stack overflow occurs in the system, it must be protected to ensure normal operation. IRePf proposes a stack overflow detection method. This method directly returns the stack data from the backup stack. By modifying the thread’s stack base address, it points to the backup stack. When the RET instruction returns, the system returns directly from the backup stack and discards the corrupted stack space.

6. Experiment and Analysis of Results

6.1. Experimental Construction. To verify the effectiveness of IRePf stack overflow detection, it is verified from the aspects of stack overflow detection capability, system performance, and backup stack overflow.

Defense Capability. The experimental environment is configured as follows: Intel Core(TM) processor, 2.6 GHz, 16 GB, Windows 10 operating system, VS2013. The designed experiments are mainly aimed at detecting two typical stack overflow attacks that target the overriding return address EIP and the current EBP of the calling function. The experiment mainly includes three parts: IRePf, simulated stack overflow attack, and upper-layer interaction system. IRePf is packaged as a kernel driver and deployed in the operating system, which is loaded at system startup. The simulated stack overflow attack is encapsulated as a direct-running stack overflow attack program and loaded into the system where IRePf is deployed. The upper-layer interaction system realizes the interaction between the user layer and the kernel layer. When a stack overflow attack occurs, the upper-layer interactive system will display relevant messages on the system desktop, including the attack type, attack mode, and attack target.

6.2. IRePf-Based Stack Overflow Detection Verification. In this article, we have written a driver A that uses stack overflow to modify the return address EIP of a function and a driver B that uses stack overflow to modify the current EBP of the calling function. These two drivers are encapsulated into stack overflow attack simulators A and B, which can be run directly. To reflect the randomness of stack overflow attack instances, a random function is inserted into A and B to generate different attack instances to prevent unexpected factors and ensure system capability. The results of running simulator A or B on a system without IRePf loaded are shown in Figure 12.

In this article, we run simulators A and B on a system loaded with IRePf. During the experiment, simulators A and B were randomly run 100 times, the stack overflow detection rate of IRePf was 100%, and a prompt message of “stack overflow attack found” popped up in the lower corner of the system desktop, including the type of stack overflow attack, attack mode, and attack target. To verify the long-running capability, we tripled the number of runs and loaded some conventional programs at the same time. IRePf still completed the discovery of all stack overflow detections.

In the analysis of results, based on the above experimental results, running simulator A aimed at modifying the return address EIP of the function, and simulator B aimed at modifying the current EBP of the calling function on the system without IRePf loaded, resulting in a system blue screen, verifying the written effectiveness of simulation program A or B. When running simulation programs A and B on the system loaded with IRePf, the system completed the detection of all stack overflow attacks. The results show that IRePf has the ability to detect attacks targeting EIP and EBP.

To further verify the detection capability of IRePf, 11 known stack overflow vulnerabilities were selected for testing. The test results are shown in Table 1. As seen from the table, the stack overflow detection rate of IRePf is still 100%.

6.3. Performance Verification of IRePf. To verify that IRePf has less system overhead, we compare the system resources (including CPU usage and memory usage) without IRePf
A problem has been detected and Windows has been shut down to prevent damage to your computer.

```
PAGE_FAULT_IN_NONPAGED_AREA
```

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software, disable memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select advanced Startup Options, and then select Safe Mode.

Technical information:

```
** STOP: 0x00000050 (0x80000000,0x00000001,0x83FF27F0,0x00000002)
```

Collecting data for crash dump ...

** Initializing disk for crash dump ...**

---

**Figure 12:** Running simulator A or B on a system without IRePf.

**Table 1:** The detection results of stack overflow vulnerabilities.

<table>
<thead>
<tr>
<th>Appliance</th>
<th>Vulnerability number</th>
<th>Detection</th>
<th>Defense</th>
<th>Attack target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kingsoft Writer</td>
<td>CVE-2013-3934</td>
<td>✓</td>
<td>✓</td>
<td>Return address</td>
</tr>
<tr>
<td>KingsoftWPSOffice wpsio.dll</td>
<td>CVE-2012-4886</td>
<td>✓</td>
<td>✓</td>
<td>Return address</td>
</tr>
<tr>
<td>Xfig</td>
<td>CVE-2010-4262</td>
<td>✓</td>
<td>✓</td>
<td>Return address</td>
</tr>
<tr>
<td>White_Dune “Scene.cpp”</td>
<td>CVE-2008-0100</td>
<td>✓</td>
<td>✓</td>
<td>EBP</td>
</tr>
<tr>
<td>VCDGear CUE</td>
<td>CVE-2007-2568</td>
<td>✓</td>
<td>✓</td>
<td>Return address</td>
</tr>
<tr>
<td>Genann</td>
<td>CVE-2018-13996</td>
<td>✓</td>
<td>✓</td>
<td>EBP</td>
</tr>
<tr>
<td>PHP</td>
<td>CVE-2018-7584</td>
<td>✓</td>
<td>✓</td>
<td>Return address</td>
</tr>
<tr>
<td>Artifex MuPDF</td>
<td>CVE-2018-1000038</td>
<td>✓</td>
<td>✓</td>
<td>Return address</td>
</tr>
<tr>
<td>Long Range ZIP</td>
<td>CVE-2017-9929</td>
<td>✓</td>
<td>✓</td>
<td>Return address</td>
</tr>
<tr>
<td>ERDAS ER Viewer</td>
<td>CVE-2013-0726</td>
<td>✓</td>
<td>✓</td>
<td>EBP</td>
</tr>
<tr>
<td>IconCool MP3 WAV Converter</td>
<td>CNNVD-201303-507</td>
<td>✓</td>
<td>✓</td>
<td>EBP</td>
</tr>
</tbody>
</table>

---

**Figure 13:** The change trend of CPU occupancy in 5 minutes.
and with IRePf. The experiments were divided into two groups. A group of IRePf-loaded and non-IRePf-loaded systems monitored CPU and memory usage over a 5-minute period to test performance. Another group measured the CPU usage and memory usage of the systems with and without IRePf loaded for 10 minutes to test the performance.

Figures 13 and 14 show the CPU usage and memory usage of IRePf-loaded and non-IRePf-loaded systems over 5 minutes. Each node is taken as the acquisition period of 15 seconds, and the average value of 15 seconds is taken as the value of each node. The horizontal axis represents the running time, and the vertical axis represents the CPU usage and memory usage.

Figures 15 and 16 show the CPU usage and memory usage of IRePf-loaded and non-IRePf-loaded systems over 10 minutes. Each node is taken as the acquisition period of 30 seconds, and the average value of 30 seconds is taken as the value of each node. The horizontal axis represents the running time, and the vertical axis represents the CPU usage and memory usage.

From the experimental results in Figures 13–16, the average CPU and memory occupancy rates of the operating systems with and without IRePf loaded can be obtained. The results are shown in Table 2.

From Figures 13–16, it can be seen that the CPU usage and memory usage of the system loaded with IRePf are
higher than those of the system without IRePf loaded. Among them, the CPU usage is approximately 5% higher, and the memory usage is approximately 100 M higher. This change will not affect the normal operation of other programs in the system; fewer resources are utilized, and the consumption of CPU and memory tends to remain balanced over time.

From the analysis in Table 2, it can be found that the average CPU usage of the two groups of experimental data of the system after loading IRePf is 9.2% and 9.3%, values which are only 4.6% and 4.7% higher than the CPU usage without IRePf. The average memory usage after loading IRePf is only 269.6 M and 269.5 M, respectively, values which are only 101.2 M and 101.1 M higher than the average memory usage without IRePf. This change will not affect the normal operation of other programs in the system, and it occupies fewer resources. As time goes by, the consumption of CPU and memory tends to remain balanced, which confers a certain average performance advantage. Therefore, this is a practical and efficient stack overflow detection technique. The existing stack overflow detection technology mainly realized by extending the compiler and disassembly technology. Stack overflow detection technology based on extended compilers has a limited scope of use, and different fields have specific requirements for compiler types and usage, so it does not have wide applicability in applications, while disassembly-based technology is mostly theoretical; there are few practical detection tools, for example, the currently highly cited dynamic detection tool StackGuard [27] and the improved StackShield [28] and PointGuard detection tools. The kernel from the system startup is used to realize the comparison between the stack overflow detection tool and the IRePf system. This is unique for the time being, so the experimental methods in this paper are not compared and analyzed. At the same time, in view of the slightly high CPU and memory usage in this article, we can encapsulate the same set of functions of the program into a class when the system is running. Due to the characteristics of the class, the memory usage will be reduced; if the program runs all the time, it can be made into a tray icon, which will save a lot of memory, and its minimization can also reduce the occupied resources and reduce the priority of the process; after the subroutine is called, the return command is used to release resources to reduce CPU and memory consumption.

6.4. Verification of the Stack Overflow Defense Capability of the Backup Stack. To verify that the stack overflow defense capability of the backup stack can defend against stack overflow attacks, we use the previously written drivers A and B to run multiple stack overflow attack simulators A and B on the system loaded with IRePf. During the experiment, after renaming simulators A and B as stack overflow attacks,
they ran 100 random times. The stack overflow defense rate of IRePF was 100%, and a prompt message of “stack overflow defense attack succeeded” popped up in the lower corner of the system desktop, including information on stack overflow attack types, attack modes, and attack targets. In the actual system application, stack overflow detection is completed immediately, and the defense is performed immediately. Overflow detection and defense are continuous and integrated implementation processes, which are specifically explained here.

Through the above experimental results, it can be shown that the established IRePF stack overflow detection platform can effectively detect and defend against stack overflow attacks that target modifying the return address EIP of the function and modifying the current EBP of the calling function. This verifies the effectiveness of the proposed IRePF stack overflow detection defense technique.

7. Conclusions and Future Work

This paper proposes an instruction reorganization virtual platform, IRePF, for kernel stack overflow detection. IRePF monitors call and RET instructions in real time through dynamic instruction reorganization, including monitoring the creation and termination of threads and the establishment and destruction of backup stacks. It not only monitors the return address but also solves the problem of unbalanced function calls and returns, and the monitoring results can determine whether a stack overflow attack occurs. The experimental results show that IRePF can effectively detect stack overflow attacks, has low system resource occupancy and high real-time performance, and effectively improves the ability and security of defense stack attacks.

To date, many stack overflow detection techniques mostly remain at the level of theoretical research and are not practical. The IRePF technology proposed in this paper, an instruction reorganization virtual platform for kernel stack overflow detection, is practical and effective, can be loaded into any operating system, such as Windows, Linux, Android, and other operating systems, and has wide applicability. Looking forward to the future, with the continuous deepening of machine learning, the features of the training set will be extracted through training and combined with contextual semantics to reduce the problem of false negatives and false positives in stack overflow, as well as to enable the comprehensive use of static detection and dynamic detection technology. This paper does not cover research on more efficient and practical stack overflow detection, and defense technology will be the goal of our future efforts.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was partially supported by Major Special Fund Projects in Heilongjiang Province, China (Grant no. 2020-230100-54-01-000358).

References


