HyperCrop: A Hypervisor-Based Countermeasure for Return Oriented Programming*

Jun Jiang[1], Xiaoqi Jia[1], Dengguo Feng[1], Shengzhi Zhang[2], and Peng Liu[2]

[1] State Key Laboratory of Information Security,
Institute of Software, Chinese Academy of Sciences,
Beijing 100190, China
{jiangjun,xjia,feng}@is.iscas.ac.cn
[2] Pennsylvania State University,
University Park, PA 16802, USA
suz116@psu.edu, pliu@ist.psu.edu

Abstract. Return oriented programming (ROP) has recently caught great attention of both academia and industry. It reuses existing binary code instead of injecting its own code and is able to perform arbitrary computation due to its Turing-completeness. Hence, It can successfully bypass state-of-the-art code integrity mechanisms such as NICKLE and SecVisor. In this paper, we present HyperCrop, a hypervisor-based approach to counter such attacks. Since ROP attackers extract short instruction sequences ending in `ret` called "gadgets" and craft stack content to "chain" these gadgets together, our method recognizes that the key characteristics of ROP is to fill the stack with plenty of addresses that are within the range of libraries (e.g. libc). Accordingly, we inspect the content of the stack to see if a potential ROP attack exists. We have implemented a proof-of-concept system based on the open source Xen hypervisor. The evaluation results exhibit that our solution is effective and efficient.

Keywords: Return oriented programming, Hypervisor-based security, Hardware assisted virtualization.

1 Introduction

Recently, return oriented programming (ROP) has drawn a lot of researchers' attention [18]. ROP attacks stem from previous return-to-libc (RTL) attacks [20]. Generally, RTL attacks construct a stack via techniques, such as buffer overflow, to transfer the control flow of a program to a function of libc (e.g., system) with provided parameters (e.g., "/home/user/evil") to perform function

* This work was supported by National Natural Science Foundation of China (NSFC) under Grant No. 61100228 and 61073179. Peng Liu was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), and NSF CNS-0905131.

S. Qing et al. (Eds.): ICICS 2011, LNCS 7043, pp. 360–373, 2011.

calls from the victim program. ROP attacks refine this idea and use a stack to redirect the control flow to a specific location inside a library, rather than the entry of a function. The location is carefully chosen so that the attacker can retain the control of the execution flow. Specifically, the attacker firstly identifies instruction sequences that end in `ret` instruction, which are called "gadgets". Due to the large code base provided by libraries installed on a common computer, theses gadgets can constitute a Turing-complete gadget set. The attacker can select several useful gadgets to perform desired computations. Since each gadget ends in `ret`, the attacker can exploit the buffer overflow vulnerability of the victim program and fill the stack with corresponding addresses. We will illustrate the ROP attacks by an example in Section 2.1.

The key point of ROP attacks is that no code injection is needed. Instead, the existing legitimate library code is misused intentionally to perform malicious behavior. Hence, many code integrity mechanisms such as NICKLE [15] and SecVisor [17] are not designed to protect the system against such attacks. Besides, libraries leveraged by ROP attacks are also essential to benign programs, so it's not feasible to counter such attacks by removing these libraries. Moreover, though the ROP attack was first introduced on the x86 architecture, now it has been extended to various architectures such as ARM [9] and SPARC [4]. Featuring these characteristics, ROP has become a new threat to computer security.

Several solutions for ROP have been proposed by researchers and can be categorized in instrumentation-based solutions [5,6] and compiler-based solutions [10,14]. However, these existing countermeasures suffer from various disadvantages and practical deficiencies. We will discuss the details in Section 7.

In this paper, we present HyperCrop, a hypervisor-based approach for ROP detection and prevention. We summarize the key characteristic of ROP attacks, and perform the stack content inspection method to calculate library addresses within the stacks and to protect the target system from the attacks. We also leverage the contemporary hardware virtualization technique, i.e., Intel VT, to facilitate our design.

In summary, we make the following contributions:

- We propose a hypervisor-based approach to defend ROP attacks. To the best of our knowledge, we are the first to use hardware-assisted virtualization as the underlying technology for ROP defense. Our approach is transparent to the protected system and does not rely on its correctness.
- We present a novel algorithm to detect ROP attacks by inspecting the content of current thread stack. By adjusting the parameters, this algorithm can effectively detect ROP attacks and has low false alarm rate.
- We have implemented a proof-of-concept system called HyperCrop based on the open source Xen hypervisor [2]. Our evaluation shows that this system is effective and efficient.

The rest of this paper is organized as follows. In section 2, we discuss the attack scenario and threat model of ROP attacks. Our system design and implementation are presented in Section 3 and Section 4, respectively. Then we give the

evaluation results in Section 5. After that, possible limitations and future work are discussed in Section 6. Finally, we discuss related work in Section 7 and conclude our paper in Section 8.

2 Attack Scenario and Threat Model

In this section, we first give a typical example to show how ROP attacks are constructed and applied. Then we identify the key feature of ROP attacks and take advantage of it for defense. Finally, we generalize a threat model of ROP and make some assumptions in our countermeasure.

2.1 ROP Attack Scenario

We demonstrate how a ROP attack is constructed. First, we can consider how one instruction is written in a return-oriented way. Take `inc eax` as an example, the attacker first obtains that its machine code is `0x40`, and appends it with a `ret` whose machine code is `0xc3`. Then he searches the byte sequence `0x40c3` in a library, and obtains the location of the sequence. This location is translated into a runtime memory address, which is finally put on top of the stack. Since complicated functionality is composed by multiple instructions, the attacker can use the above method to process instructions one by one to accomplish a real attack. Note this can be done totally in an automatic way [7]. When the payload is generated, the attacker feeds it to a program with buffer overflow vulnerability. On top of the stack resides the address of the first instruction to be executed, and a simple `ret` will trigger the attack.

Several observations are presented below. Firstly, some buffer overflow vulnerabilities are caused by `strcpy`, so null byte should be avoided in the payload. Secondly, it is difficult to find a gadget corresponding to a complicated instruction whose machine code is really long, so such instruction should be decomposed into several smaller ones. Finally, immediate data can also be put on the stack. Specific details can be found in [16].

From above we summarize that the key characteristic of the ROP attack is filling the stack with a payload containing plenty of addresses that are within libraries and a few immediate data. Normally, the stack is used for local variables, function parameters and return addresses. We believe that these two kinds of stack usage can be distinguished with reasonable effort, which is the key of our countermeasure.

2.2 Threat Model and Assumption

Inspired by the above scenario, we generalize the threat model of ROP attacks. We assume that there are programs in the protected system that are vulnerable to stack overflow attack. We consider those programs to be our protection targets. The attacker tries to exploit such overflow vulnerability to perform ROP attacks.

Since our method involves virtualization technology, we also make the assumption that the VMM is always trusted. This is usually a fundamental assumption for hypervisor-based security researches and is consolidated (though not guaranteed) by existing hypervisor protection mechanisms [12,1,19]. In our threat model, we assume that the VMM is secure and the attacks to the VMM are out of the scope of this paper.

3 System Design

In this work, we design a hypervisor-based approach to defend ROP attacks. Specifically, we utilize the higher privilege of the VMM to intercept write operations to the stack and to inspect the content of the stack for the detection of ROP attacks. Moreover, our system should work effectively and introduce as low performance overhead as possible.

Our system is built upon the existing VMM and contains four parts: an event handling component, a stack marking component, a breakpoint locating component and a stack inspecting component. The system architecture is shown in Fig. 1.

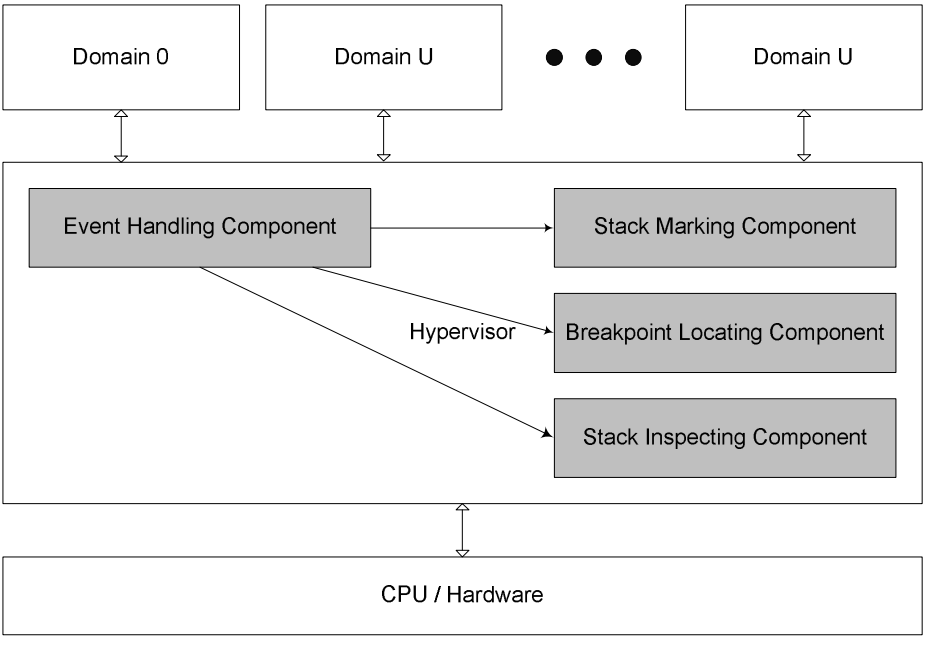

Fig. 1. Architecture of HyperCrop

3.1 Event Handling Component

Since we use hypervisor based approach to defend ROP attacks, in order to intercept stack write and perform stack content inspection correctly, we need to modify the following three event handlers of the VMM.

Context Switch Handler. Because our protection target is a single program, we limit our scope to a specific process. We modify the context switch handler of the VMM to identify if the next-to-be-scheduled process is our protection target. If so, the stack marking component is called to mark the stack of the process read-only in the shadow page table. Detailed explanations are given in Section 3.2.

Page Fault Handler. The page fault handler of the VMM needs to be modified in order to intercept write operations to the stack and call our breakpoint locating component. Since the stack is marked as read-only, a write to the stack will cause a page fault and will be trapped by the VMM. The default page fault handler will notice that this page fault is caused by intentionally marking a writable memory region as read-only, so it will mark the corresponding stack area writable again. Our additional handler will then call the breakpoint locating component to find a specific position as a hardware breakpoint. The selection method of the breakpoint is discussed in Section 3.3.

Debug Exception Handler. The hardware breakpoint will trigger a debug exception, which is also trapped by the VMM. We modify the handler to perform two operations: a) mark the stack area read-only again; b) call the stack inspecting component to detect whether a ROP attack is present.

3.2 Stack Marking Component

This component will be used under two circumstances: a) when a context switch happens and the next-to-be-scheduled process is our protection target; b) when a stack inspection is performed. The purpose of marking stack as read-only is to arouse page fault so that write operations to the stack can be intercepted and processed.

Under x86 architecture, the top of a stack is designated by the general purpose register `esp`. However, as the program executes, its value changes frequently. Since modern operating systems usually allocate fixed memory space for the stack, stack range can be obtained from the OS data structures and the corresponding memory pages can be marked as read-only. Under virtualization environment, shadow paging is a common software solution for memory virtualization. When shadow paging is utilized in virtualization environment, a shadow page table is used for address translation, so its entries should be set as non-writable.

3.3 Breakpoint Locating Component

A breakpoint is a checkpoint where inspection of the stack content should be performed. This component is called by the page fault handler, and the value provided by this component will be used to set as the breakpoint.

Since a ROP attack is initiated by a `ret` instruction, this component will find the next `ret` in the instruction stream from current `eip`. Instructions between current `eip` and the breakpoint will run natively and will not be disturbed. This is an important performance guarantee of our system since the stack is used very frequently. Inspecting the content of the stack every time when it is written would introduce significant performance overhead.

3.4 Stack Inspecting Component

When a breakpoint is met, HyperCrop performs stack content inspection in order to identify whether a ROP attack is present.

According to the feature summarized in Section 2.1, we devise the following algorithm to inspect the content of the stack.

Definition 1. *Potential Gadget Address Set (PGAS) is a set of addresses that are within the address space of the libraries (e.g., libc) which attackers could potentially exploit. We use Φ to represent PGAS.*

Definition 2. *Top N Stack Element Array (TNSEA) is an array of elements that is located on top of the stack. Each element is 4-byte long for alignment, and the stack grows down. So the TNSEA occupies the virtual address space ranging from [esp] to [esp] - 4 * N. We use $\mathcal{A}$ to represent TNSEA, and $\mathcal{A}[i]$ to represent the i-th element of TNSEA, which is located at [esp] - 4 * (i - 1).*

Definition 3. *Suspected Gadget Address Ratio (SGAR) is a value indicating the number of elements in TNSEA which belong to PGAS. We use R(N) to represent SGAR, where N is the size of TNSEA. So we have* $R(N) = \frac{\sum_{i=1}^{N} (\mathcal{A}[i] \in \Phi) ? 1 : 0}{N}$. *Obviously, the range of R(N) is from 0 to 1.*

Since attackers fill the stack with plenty of return addresses, the values returned from function R will be much larger (e.g., > 0.8) if a ROP attack is present. We may set up a threshold and compare it with the calculated value R to identify such attacks. The threshold must be carefully chosen in order to balance the false positive rate and the false negative rate.

4 Implementation

We have developed a proof-of-concept system based on Xen 4.0.1, running fully-virtualized Windows XP SP3 (x86-32) on top of a 64-bit VMM. Our development machine has an Intel Core i5 processor with the latest hardware virtualization technology support. In the following, we present some implementation details for the key techniques in our approach.

4.1 Modifications to Xen's Handlers

In hardware-assisted virtualization environment, if a guest OS tries to perform some privileged or sensitive operations, an event called VMEXIT will occur. The processor switches its privilege level from VMX non-root mode to VMX root mode, and calls a previously registered VMEXIT handler to deal with this event. Xen handles these events appropriately for correct virtualization, and we make several modifications to fulfill our needs.

Modifications to the Context Switch Handler. The context switch mechanism provided by the x86 architecture called TSS (Task State Segment) is not used by modern operating systems. Instead, we find that a context switch involves the change of the page table base address register (i.e., `CR3`), because modern operating systems utilize the paging mechanism to achieve process space isolation. Therefore, our modification target is `CR_ACCESS` instead of `TASK_SWITCH`. Moreover, the stack area of a certain thread is fixed. We locate the range of the stack through the kernel data structure information every time when a context switch occurs and save it for future use.

Modifications to the Page Fault Handler. The page fault handler is one of the most complicated parts of the VMEXIT handler under shadow paging mode. We write our own page fault handler to wrap the original one. The return value of the page fault handler is either "1" or "0". The former indicates that the page fault is caused by the "out of sync" shadow page table, while the latter indicates that this is a real guest page fault and the operating system should handle it. So we define a page fault is caused by stack write if the following three conditions are satisfied: a) the original page fault handler returns "1"; b) the page fault linear address is within the stack range; c) the page fault error code indicates a write access. For stack writes, an appropriate hardware breakpoint is set via x86 debugging mechanism. The `DR0` register is used to save this breakpoint and relevant bits in `DR7` are set to enable this breakpoint.

Modifications to the Debug Exception Handler. The original debug exception handler is a part of Xen's debugging facility. We modify the handler to implement our functionality. First of all, the original handler crashes a domain if a debugger is not attached or the monitor flag is not set. We remove this restriction to prevent the domain crashing. Moreover, we clear the content of the DR0 register to disable the current breakpoint. Finally, we trigger the stack inspecting component to identify potential ROP attacks and stack marking component to mark the stack as read-only again so a later stack write will cause a new inspection loop.

4.2 Marking the Stack

To mark the stack as read-only, we utilize several shadow page table related functionalities provided by Xen hypervisor. Firstly we walk through the shadow

page table for the specific linear address. We obtain the corresponding page table entry from the walk-through results and reset the RW bit. Then the new page table entry is written to the shadow page table. Finally a TLB flush is performed since the access privilege is lowered.

4.3 Locating Breakpoints

In order to locate a breakpoint where stack content inspection needs to be performed, we disassemble the instructions from current `eip` to find the next `ret` instruction to be executed. Furthermore, we are only concerned about the length of the instructions instead of exact disassembling results. Thus, we only need to identify the opcode, which determines the addressing mode and ultimately the instruction length. Since the implementation is inside Xen kernel without any method to resort to third party libraries, we have to write the disassembling code by our own.

The instruction complexity of x86 (a kind of CISC architecture) imposes several challenges to our implementation. Moreover, besides `ret`, there are other control transfer related instructions whose execution may diverge the control flow:

- Conditional jumps. Since Xen does not emulate the execution of the instructions, we cannot calculate whether the instructions before conditional jumps have effects on the EFLAGS register.
- Jumps and calls whose targets are given by registers or memory locations. Similarly, we do not calculate whether the instructions before them change the value of the specific register or memory location. Hence, we do not determine these jumps/calls' targets. Instead, we only handle conditional jumps and calls whose target is specified in the form of immediate number.
- Instructions that cause privilege level change. These instructions cause control flow transfer as well as privilege level change. Since we currently do not trace into kernel, we ignore these instructions.
- Some complicated instructions. These instructions include infrequently used 2-byte-opcode instructions, e.g., all MMX, SSE, VMX instructions. Actually they are rarely used in our experimental instruction sequences. So we think not handling them will not affect the effectiveness and performance.

When we encounter the above instructions, we consider them to be the breakpoint locations as well as `ret`.

4.4 Inspecting the Stack Content

In order to inspect the content of the stack, we first read the stack elements by copying 400 bytes from `esp` (since we inspect the top 100 elements). For each element, we check if it is within the range of libraries. If so, we increase a counter α. Finally, we calculate the ration $\rho = \frac{\alpha}{100}$. In our implementation, we find that sometimes the `esp` points to the kernel stack instead of the user stack. Since the kernel stack is not of our concern, we ignore such cases.

5 Evaluation

In this section, we give the experimental evaluation of our HyperCrop prototype. Our evaluation has two goals. The first is to evaluate the effectiveness of HyperCrop for defending real ROP attacks, while the second is to measure the performance overhead introduced by the system using benchmarks.

The following experiments were all conducted on a machine with Intel Core i5-760 processor and 8GB memory. The version of Xen used in our experiment is 4.0.1 and the dom0 is 64 bit CentOS 5.5 with kernel version 2.6.34.4. The guest OS is Windows XP SP3 allocated with one processor core and 2 GB memory.

5.1 Effectiveness

In order to evaluate the effectiveness of our system, we first recall the characteristics of the ROP attack we summarized in Section 2.1. The stack contains many return addresses within the libraries, which can be easily distinguished from normal stacks. We consider this anomaly by previously defined function R(N). Specifically, we run 12 normal programs and calculate the results of the function R(N). These results are corresponding to "normal" stacks. We select the DLL memory regions which are related to the protected program such as 0x7d590000 ~ 0x7dd84000 (shell32.dll), 0x7c800000 ~ 0x7c91e000 (kernel32.dll), 0x7c920000 ~ 0x7c9b3000 (ntdll.dll) and 0x77d10000 ~ 0x77da0000 (user32.dll)[1]. These four DLLs are similar to libc since they are dynamically linked to almost every executable.

Table 1 shows that the results of function R(N) are relatively small for normal programs. The above four DLLs are selected because the attacker usually hopes that the code base is general to all programs. Some attackers may specifically analyze a certain program for exploitation, hence all the DLLs linked by the program can be available as code base. We also conducted such experiment by selecting 0x20000000 ~ 0x7ffd0000[2] as the range. This range is big enough to contain almost all DLLs linked by a program (occupying almost 3/4 of the user space). The corresponding Max R(100) and Average R(100) are 56% and 19%, respectively. In contrast, a typical ROP attack payload usually contains tens or hundreds of return addresses and a few immediate data. There does exist a manifest boundary between normal stacks and attacked stacks. Hence, we believe that our system provides an effective countermeasure for ROP attacks. However, several elaborate attacks may affect our inspection algorithm as discussed in Section 6.

[1] We parse the PE structure manually to obtain these memory ranges. Windows XP does not support ASLR (address space layout randomization) so these addresses are fixed. On operating systems that support ASLR, these addresses can be obtained by traversing the VAD (virtual address descriptor).

[2] 0x7ffd0000U to 0x7fffffffU are used for PEB and TEB, so we don't include this area to avoid false alarms.

Table 1. Result of function R(N) for normal stacks

Program	Inspection Count	Max R(100)	Average R(100)
Notepad	3679	24%	5.5%
MS Paint	3730	33%	8.4%
Calculator	2507	27%	5.1%
Mine Sweeper	953	24%	10.0%
RegEdit	616	22%	8.1%
7-Zip	307	27%	8.2%
WinRAR	960	21%	5.8%
Internet Explorer	8140	27%	5.1%
Mozilla Firefox	6899	30%	3.6%
Microsoft Word	9302	31%	4.6%
Microsoft Excel	15605	32%	5.7%
Microsoft PowerPoint	6983	29%	6.5%

5.2 Performance

In order to evaluate the performance of our system, we use DAMN Hash Calculator and Everest to perform several benchmark tests. We show the normalized performance overhead in Fig. 2. The evaluation results demonstrate that our system incurs acceptable performance overhead and provides a practical countermeasure for ROP attacks.

6 Discussion

The above evaluation exhibits that our current prototype can successfully and practically defend ROP attacks. In this section, we discuss several limitations of HyperCrop prototype.

Firstly, several researchers have put forward other relevant attacks. Instead of `ret` instructions, attackers can use other control flow transfer instructions like `jmp` to initiate similar attacks and compromise the control flow [3]. Since these attacks do not rely on the stack for control flow retention, our countermeasure cannot defeat such attacks. Moreover, ROP attacks can also be mounted on the operating system kernel [7], while our system currently protects user applications only. We leave defense measures for these new attacks as our future work.

Secondly, the hardware-assisted virtualization is currently only available for x86 architecture, limiting its application for other architectures. Software-based virtualization is able to solve this problem though it may introduce more overhead. We believe hardware-assisted virtualization for other platforms will be available as the VM techniques develop, then our approach can be extended to those platforms.

Finally, our algorithm may be subverted by some tricks, e.g., replacing `ret` instruction with `retn` instruction. The `retn` instruction has an immediate parameter. It not only pops the top of stack to `eip` but also changes the value of `esp`.

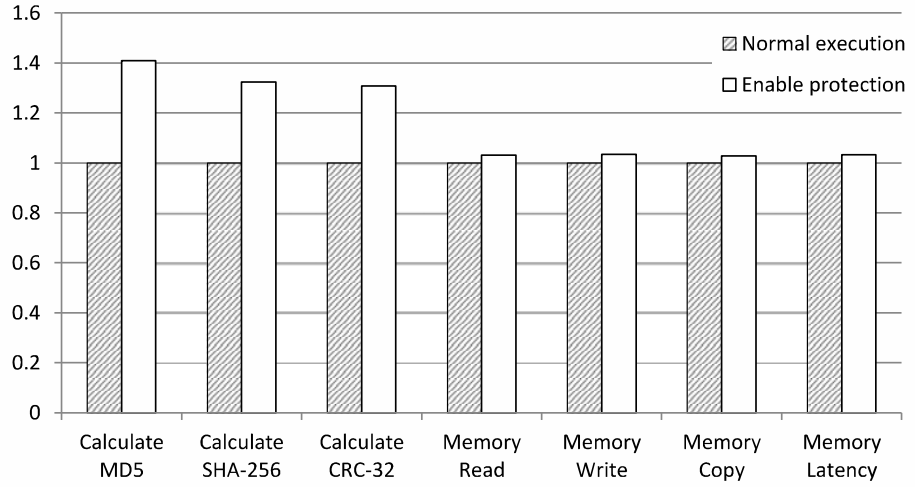

Fig. 2. Normalized performance overhead

Hence, the return addresses in the stack can be separated by junk data, which makes the stack resemble normal. However, we don't think such attack is a big threat to our system. First, `retn` instruction is not frequently used in libraries making it difficult to generate a Turing-complete gadget set with `retn`. Second, to subvert our approach, the return addresses must be distributed sparsely enough within the stack. However, the stack size of the system is limited, and performing a task in a return-oriented way needs more instructions (return addresses) than normal. These constraints make it difficult for the attackers to use `retn` in their attacks. Finally, we can enhance our inspection algorithm. For example, we can use approaches similar to program shepherding [8] for fine grained ROP detection or integrating existing ROP defending techniques in our framework [5,6].

7 Related Work

W^X Mechanism. W^X mechanism [21] is an effective countermeasure to code injection attacks, but not for ROP attacks. An attacker utilizes vulnerabilities like stack overflow to inject his or her own code to the stack of the victim program. A key feature of code injection is that the code is written to memory as data but executed as code. The W^X mechanism ensures that the writable memory region must not be executable, hence naturally defeats code injection attacks. However, the data written by ROP attacks to the stack are return addresses and immediate data, which are never executed as code. Hence, ROP attacks can succeed without violation of W^X policy.

ROP Defense. There are several existing solutions for ROP attacks, relying on different characteristics of such attacks. On one hand, since ROP attacks rely on the `ret` instructions of the library files, an intuitive idea is to remove all `ret` instructions, both intentional and unintentional, and to use return indices to replace return addresses in the stack to avoid pop-jmp scheme which is also exploitable to attackers [10]. The approach proposed in [14] is more deliberate and eliminates almost all possibilities that a benign library could be misused. These compiler-based solutions require the availability of the library source code, hence are not able to protect commercial operating systems (e.g. Microsoft Windows) against such attacks. On the other hand, a key feature of ROP attacks is frequent appearance of `ret` in the instruction sequence [5]. Furthermore, these `ret` instructions are deliberately crafted and have no corresponding `call`s, so the stack's First-In-Last-Out property is violated [6]. Hence several researchers use dynamic binary instrumentation frameworks like Pin [11] or Valgrind [13] to instrument program code to identify such attacks. These instrumentation-based solutions incur high overhead (e.g., $> 400\%$). Compared to those kinds of methods, HyperCrop does not require source code and has low performance penalty.

8 Conclusion

In this paper, we present HyperCrop, a hypervisor-based system for defending ROP attacks. We leverage the contemporary hardware assisted virtualization technology as well as novel algorithm design to achieve high detection rate and low performance overhead. Our experiments exhibit that HyperCrop can effectively and efficiently detect ROP attacks before their initiation. We believe that our system provides a practical defense again ROP attacks.

References

1. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 38–49. ACM, New York (2010)
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 164–177. ACM, New York (2003)
3. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, pp. 30–40. ACM, New York (2011)
4. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 27–38. ACM, New York (2008)

5. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: Detecting Return-Oriented Programming Malicious Code. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 163–177. Springer, Heidelberg (2009)
6. Davi, L., Sadeghi, A.-R., Winandy, M.: ROPdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, pp. 40–51. ACM, New York (2011)
7. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM 2009, pp. 383–398. USENIX Association, Berkeley (2009)
8. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, pp. 191–206. USENIX Association, Berkeley (2002)
9. Kornau, T.: Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universitat Bochum (2010)
10. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with "return-less" kernels. In: Proceedings of the 5th European Conference on Computer Systems, EuroSys 2010, pp. 195–208. ACM, New York (2010)
11. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005)
12. Murray, D.G., Milos, G., Hand, S.: Improving Xen security through disaggregation. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2008, pp. 151–160. ACM, New York (2008)
13. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 89–100. ACM, New York (2007)
14. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-Free: defeating return-oriented programming through gadget-less binaries. In: Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC 2010, pp. 49–58. ACM, New York (2010)
15. Riley, R., Jiang, X., Xu, D.: Guest-transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
16. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. (to appear, 2011)
17. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 335–350. ACM, New York (2007)
18. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 552–561. ACM, New York (2007)

19. Wang, Z., Jiang, X.: HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 380–395. IEEE Computer Society, Washington, DC, USA (2010)
20. Wikipedia. Return-to-libc attack (2010), http://en.wikipedia.org/wiki/Return-to-libc_attack
21. Wikipedia. W^X (2010), http://en.wikipedia.org/wiki/W%5EX