energies

MDPI

Article

Lightweight and Seamless Memory Randomization for Mission-Critical Services in a Cloud Platform

Joobeom Yun [1,†], Ki-Woong Park [1], Dongyoung Koo [2,*] and Youngjoo Shin [3,*]

[1] Department of Computer and Information Security, Sejong University, Seoul 05006, Korea; jbyun@sejong.ac.kr (J.Y.); woongbak@sejong.ac.kr (K.-W.P.)
[2] Department of Electronics and Information Engineering, Hansung University, Seoul 02876, Korea
[3] Department of Computer and Information Engineering, Kwangwoon University, Seoul 01897, Korea
* Correspondence: dykoo@hansung.ac.kr (D.K.); yjshin@kw.ac.kr (Y.S.); Tel.: +82-2-760-5751 (D.K.); +82-2-940-5130 (Y.S.)
† Current address: Rm. #724, Daeyang AI center, Seoul 05006, Korea.

Received: 3 February 2020; Accepted: 9 March 2020; Published: 13 March 2020

Abstract: Nowadays, various computing services are often hosted on cloud platforms for their availability and cost effectiveness. However, such services are frequently exposed to vulnerabilities. Therefore, many countermeasures have been invented to defend against software hacking. At the same time, more complicated attacking techniques have been created. Among them, code-reuse attacks are still an effective means of abusing software vulnerabilities. Although state-of-the-art address space layout randomization (ASLR) runtime-based solutions provide a robust way to mitigate code-reuse attacks, they have fundamental limitations; for example, the need for system modifications, and the need for recompiling source codes or restarting processes. These limitations are not appropriate for mission-critical services because a seamless operation is very important. In this paper, we propose a novel ASLR technique to provide memory rerandomization without interrupting the process execution. In addition, we describe its implementation and evaluate the results. In summary, our method provides a lightweight and seamless ASLR for critical service applications.

Keywords: address space layout randomization (ASLR); rerandomization; code-reuse attack; return-oriented programming (ROP); seamless memory randomization

1. Introduction

Nowadays, there is a lot of services being hosted on cloud computing platform. Among these services, mission-critical services such as a hypervisor also have been worked in cloud computing. However, critical vulnerabilities in cloud service application can result in huge disasters for governments, companies, and the military [1]. To prevent this threat, it is important to defend software against the vulnerability of attacks. In recent years, many studies have been aimed at developing better security solutions. Although every solution has the same goal, there are individual differences regarding the methods, and thus, pros and cons to each. Among them, the most effective methods are address space layout randomization (ASLR) [2] and data execution prevention (DEP) [3]. ASLR randomly arranges the memory regions of a process, such as text, data, stack, heap, and libraries, in order to prevent an attacker from successfully setting down onto a prepared point in the memory. DEP marks memory chunks as non-executable so that any attempt to run executable code in these chunks fails.

ASLR provides a robust way to mitigate code-reuse attacks [4]. It introduces diversity into the program binary by randomizing the memory location so as to prevent attackers from learning address information with which to mount attacks. Because its effectiveness has been proven over

the past few decades, common contemporary operating systems, such as Linux, Windows, and MacOS, are equipped with ASLR to mitigate attacks. However, the contemporary version of ASLR is inadequate in terms of meeting the desired level of security for mission-critical applications, because of its low entropy [4] and its inability to deal with more advanced and highly sophisticated attacks, such as just-in-time return-oriented programming (JIT-ROP) [5] and Blind-ROP [6]. This leads to the need for another security mechanism, one which augments the facilities and the security of the current ASLR implementations.

A mission-critical service is an application that is essential to a business operation or to an organization [7]. Its execution must not stop, either for economic reasons or because the consequences of halting the service are great. For example, supervisory control and data acquisition (SCADA) applications must not stop because they must provide seamless and crucial functions to the infrastructure. Although there are backup systems suitable for this purpose, they cause economic losses resulting from duplicate facilities or job switching. Another example is a *hypervisor* in the cloud system. The hypervisor presents guest operating systems regarding an immediate user request and manages the execution of guest operating systems, always ensuring a seamless service. In summary, it is important to protect these mission-critical services from disruptive influences.

For these applications that play a crucial role in a system, security threats are the main concern; this is because the applications are venerable to bugs that lead to arbitrary code execution, and thus give attackers full control over the system. Specifically, code-reuse attacks, such as return-oriented programming (ROP) [8], supply a vigorous and effective technique that is widely used to exploit memory corruption vulnerabilities in application software. As a consequence, it is necessary to offer strong security to mission-critical services with a high availability.

In devising a security solution for mission-critical applications, several challenging issues should be taken into consideration. First, the solution should bridge the gap between the security level offered by current ASLR implementations and the threats caused by state-of-the-art exploitation techniques, so that the target application can resist more types of code-reuse attacks. Second, the solution must not modify the underlying low-level system components, for example, the OS kernel, since access to the system is prohibited in most operating systems. Finally, the solution should assure high availability, which is a mandatory requirement for mission-critical services. This implies that neither modifying the source codes nor shutting down processes is not allowed when applying the solution to applications.

Many solutions were brought forth in the process of seeking to reinforce ASLR implementations, all of which share the same direction toward improving the granularity of binary randomization. Techniques underlying these solutions are categorized into two approaches: load-time randomization [9] and runtime randomization [10–12]; they vary in terms of the time at which the randomization occurs. In the literature, runtime randomization is accepted as the more robust solution when compared with load-time-based methods, in which a serious design flaw has been discovered [5]. State-of-the-art runtime-based solutions, however, are unfeasible when deployed in target systems (i.e., mission-critical applications) because of their fundamental limitations: the need for modifications of low-level system components [10], and the need for recompiling source codes [10] or restarting processes [11,12], which eventually incur unacceptable interruptions to the high-availability-assured target system.

In this paper, we address the aforementioned issues and propose a solution that overcomes the limitations of competitive techniques. More specifically, our solution, dubbed *SRandomizer*, cleared all the hurdles in the process of successful adaption to mission-critical applications. The underlying idea of SRandomizer is to transform a target program binary into one that is capable of repeatedly relocating itself to a random location during program execution. Also, SRandomizer gives the opportunity to introspect the memory protection of cloud computing platform.

The main contributions of this paper can be summarized as follows:

- We summarize code-reuse attacks and their countermeasures. They include the history of code-reuse attacks, and defenses, such as return-to-libc [13,14], return-oriented programming, ASLR, and DEP. In addition, we compared our solution with state-of-the-art defense solutions.
- We propose protection which can be applied to non-protected applications without modifying any system components or even suspending the application processes. Note that this is the desired property necessary to achieve high availability in mission-critical services in cloud platforms.
- We propose SRandomizer, which follows the approach of runtime rerandomization by repeating the binary relocation at extremely short intervals (i.e., 50 ms). According to Shuffler [11], a 50 ms shuffle period is enough to prevent existing JIT-ROP attacks which take 2.3 to 378 seconds to complete.
- SRandomizer is lightweight, which means that it does not require a significant overhead. Our experiment shows that SRandomizer causes a performance drop of less than 5% in the execution time of an evaluated application.
- We demonstrate that a case study (i.e., *libvirtd* which is a virtualization daemon essential for Linux cloud platform) prevent the ROP attack well. This means that SRandomizer is very useful protection against cloud platform memory attack.

The rest of the paper is organized as follows. We introduce the background in Section 2, and the approach overview and implementation of SRandomizer in Section 3. Thereafter, we evaluate the accuracy, effectiveness, and efficiency of SRandomizer in Section 4. Finally, we discuss the limitations in Section 5 and provide conclusions and future work in Section 6.

2. Background

2.1. Code-Reuse Attacks

Buffer overflows are a fatal attack approach with severe consequences for proper application reliability and consistent operation. Although many defenses against this attack have been proposed, address space layout randomization (ASLR) and data execution prevention (DEP) are the only effective ones among them [5]. DEP prohibits injected codes from executing in the memory, for instance, the stack or the heap areas, while ASLR invalidates an injected shellcode address by randomizing the memory. However, code-reuse attacks were introduced to avoid DEP. This method utilizes the program code already present in the memory because the code injection becomes difficult to use. Since the code-reuse attack uses several code snippets in the executable area, DEP protection is incapacitated against the attack. The canonical example is a return-to-libc [13,14] attack. It targets the *system()* function of the UNIX C standard library (i.e., libc) which is linked to almost every process running on a UNIX-based system. The *system()* function takes as an input a shell command to be executed. For example, the function call *system("/bin/sh")* in UNIX-based systems executes a new terminal program. This means, by invoking the *system()* function, the attacker can execute any kind of command without injecting the shellcode.

This approach was extended to return-oriented programming (ROP) [5,15], which chained together short instruction sequences ending with a *ret* instruction (called *gadgets*) so as to execute some actions (Figure 1). This control-flow attack redirects the control-flow to an unintended area; i.e., an executable library code segment. The first step to gadgets is changing the stack pointer to the beginning of the payload. A concrete example of this is *MOV rsp, rax; ret*. Once the stack pointer is moved, it supports the sequential execution of gadget payloads. These gadget payloads usually become the shellcode that permits the user to send commands. In real attacks, the attacker uses several API function pointers because it is difficult to make a perfect shellcode by using only a small piece of code. For instance, there are two functions which are frequently used to make the ROP attack more convenient and effective in the libc.so shared library. They are the *setresuid()* function, which sets real, effective, and saved user IDs; and the *system()* function, which executes the */bin/sh*. In order to find useful API functions and gadgets, an attacker scans memory layouts and discloses them. Finally, he compiles an attack using gathered API functions and gadget codes, and then executes the attack.

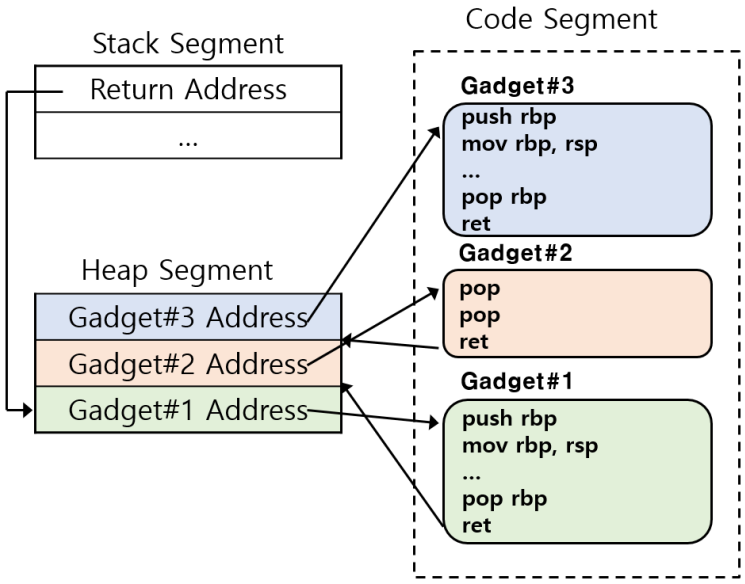

Figure 1. Typical return-oriented programming (ROP) attack.

2.2. Defenses against Code-Reuse Attacks

In order to defend against code-reuse attacks, many solutions have been proposed. First of all, control-flow integrity (CFI) [16,17] ensures that a program's control-flow follows a legitimate path in the application's control-flow graph. This CFI checks and prevents any attempt by an attacker to hijack the program's control-flow at runtime. CFI is usually implemented with a shadow stack [18,19], which implements every call and return instruction at runtime. Although CFI provides an obvious protection and does not require the source code of an application, it suffers from practical limitations, including an average performance overhead of 21% [17].

Second, there are information hiding techniques for an adversary to disclose the memory layout. Marlin [9] shuffles the internal structure of an ELF executable at load-time, giving a fine-grained level of randomization so as to effectively hinder code-reuse attacks. However, load-time randomization is susceptible to just-in-time ROP (JIT-ROP) attacks [5], whose strategy is to repeatedly abuse a memory disclosure to map an application's memory layout on the fly. In order to defend against JIT-ROP, several information hiding solutions have been proposed. Runtime randomizing approaches such as TASR [10] and Shuffler [11] try to rerandomize the memory layout of a process in the order of milliseconds so that they make the memory disclosures stale. The second defense concentrates on preventing recursive gadget harvesting. Oxymoron [12] proposed an inaccessible table to hide the true destination of *call* instructions, and Isomeron [20] randomized program execution paths. The third defense proposed an execute-only memory, either with a custom hypervisor [21] or software emulation [22]. The fourth defense [23] uses self-modifying code (SMC) technology to rewrite potentially vulnerable software binaries to chronomorphic binaries or creates new variants when it forks a child process [24].

Table 1 compares several runtime rerandomization techniques. We chose four recent rerandomization systems and compared our system with them. Isomeron [20] randomizes the program execution path and it randomizes memory layout only once. Chronomorphic [23] modifies a binary file at runtime. Isomeron and Chronomorphic require big memory overheads because they need duplicate program execution paths or a modified program copy. TASR [10] and RuntimeASLR [24] mutate the program binary or a child process, respectively. These cause unnecessary rerandomization or severe pointer tracking overheads. Meanwhile, SRandomizer only rerandomizes a shared library so that it has a small memory overhead. However, most importantly, these solutions (all except SRandomizer) are not feasible when deployed in the target systems that we are interested in because of their fundamental limitations, i.e., the need for recompiling source codes [10] or restarting processes [11,12], which eventually incur unacceptable interruptions to high-availability-assured target applications.

Stopping a mission-critical service or upgrading and updating a low-level system component requires that a critical application's service halts. This causes a loss of assured availability and results in a lack of confidence. To prevent this phenomenon, the manager backs up a secondary system, but this produces an economic loss associated with duplication. Therefore, a lightweight and seamless ASLR service is required for our target systems, which include mission-critical services.

Table 1. Comparison of runtime rerandomization techniques.

System	Trigger	Randomization	Disadvantage	Installation
Isomeron [20]	Randomizer once	Execution path	Big memory overhead	Re-execution
Chronomorphic [23]	User configuration	Program modification	Big memory overhead	Re-compile
TASR [10]	I/O calls	Program mutation	Unnecessary rerandomization	Re-compile
Shuffler [11]	Periodical	Binary and shared libraries	Binary rewriting	Re-compile
RuntimeASLR [24]	fork() call	Child process mutation	Pointer tracking overhead	Re-execution
SRandomizer (Our system)	Periodical	Shared library	Small memory overhead	Injection (No interruption)

3. SRandomizer

SRandomizer aims to prevent code-reuse attacks by rerandomizing the address space of a running process with a seamless service. More specifically, SRandomizer rerandomizes the shared memory of the critical application so that it makes the memory disclosures stale. As a consequence, it provides a seamless service in addition to invalidating an attacker's attempt.

3.1. Motivation Example

Listing 1 shows the vulnerable source code of libvirt_proxy.c in the *libvirtd* daemon, which is a hypervisor driver. This vulnerability is the libvirt_proxy buffer overflow [25]. The buffer overflow in the *proxyReadClientSocket* function in *proxy/libvirt_proxy.c* might allow a local user to gain privileges by sending the header twice: one is a part of the header of a *virProxyPacket* packet, and the other is the remainder of the packet with crafted values related to the use of uninitialized memory in the validation check. As you see in the Listing 1 code snippet, the vulnerable parts are call instructions to the C standard library (i.e., *memcpy* or *strcpy*). This is because the C standard library has buffer manipulating operations, such as copying, allocating, or freeing a buffer. In fact, most memory corruption attacks occur in C standard libraries, as is so in this example. The most effective way to invalidate the attack to the C standard library is to randomize the library so as to make the corrupted *ret* address and ROP gadgets stale.

Table 2 shows 71 shared libraries of the *libvirtd* daemon. Like many other Linux commercial on-the-shelf (COTS) programs, the KVM/QEMU hypervisor driver (i.e., *libvirtd* daemon) uses many shared libraries. Among these libraries, *libvirt.so.0* has the vulnerability described in Listing 1. An attacker can exploit this vulnerability by sending a crafted packet to the *libvirtd* daemon. Then, an ROP exploit code uses these shared libraries in order to save coding effort. At this time, it is important to prevent the ROP exploit from abusing the libraries. This protection makes code reuse ineffective, although the buffer overflow attack is successful. This is done by making the gathered code in the reusing attack stale. We will describe this in more detail in Section 3.3.

Listing 1. libvirt_proxy.c (vulnerable version).

```
static int proxyReadClientSocket(int nr)    {
virDomainDefPtr def;
virProxyFullPacket request;
virProxyPacketPtr req = (virProxyPacketPtr) &request;
... snip ...

/* Following codes are vulnerable.    */
memcpy(&request.extra.str[0], uuid, VIR_UUID_BUFLEN);
strcpy(&request.extra.str[VIR_UUID_BUFLEN], name);
... snip ...
}
```

Table 2. Shared libraries of libvirtd daemon.

No.	Shared Library	Explanation
1	libc.so.6	C standard libraries on Linux
2	libvirt.so.0	libvirt client library
3	libvirt-qemu.so.0	KVM/QEMU driver libraries
4	libpthread.so.0	Process thread libraries
5	libxml2.so.2	XML libraries
...	....	....
71	libcrypt.so.1	crypt library for DES, MD5, Blowfish, and others

3.2. The SRandomizer Approach

To make code-reuse attacks invalid, SRandomizer provides a seamless runtime rerandomization scheme for COTS programs. Conceptually, SRandomizer achieves a seamless service by randomizing the shared library without restarting the process and rerandomizes the shared library continuously during the execution. However, the construction of SRandomizer faces challenging problems: how can it achieve seamless memory randomization on a target application without (1) modifying the low-level systems and (2) interrupting the application process?

We overcame those problems by implementing SRandomizer in the form of a shared library. The shared library is executed in the user-level process context and can be injected to the memory space while a target process is running. The target process is a mission-critical application, such as an industrial system application or a hypervisor in cloud computing which requires no interruption during the execution. We protect these mission-critical applications by randomizing the essential shared library which is frequently used for code-reuse attacks in order to reduce the overhead. By rerandomizing the essential API functions (i.e., gadgets), SRandomizer makes the scanned gadgets outdated. As a consequence, an attacker cannot gather gadget codes well and the code-reuse attack fails. This is a lightweight and effective defense technique that provides seamless service to a critical application. In order to maximize the defensive effect with minimum cost, SRandomizer randomizes the C standard shared library, because that is where vulnerability usually exists; thus, it is highly probable for the attacker to gather the required code in the C standard library. However, SRandomizer can randomize whichever shared library a user wants.

3.3. Implementation

Figure 2 shows the functional overview of SRandomizer. It performs randomization in the context of a target process with its own thread, thus running concurrently with the application. Once injected

into the process, SRandomizer prepares the randomization by copying a target executable module (i.e., a shared library such as *libc.so* in Figure 2) to a memory region which is marked as *stand-by*. More specifically, the active module serves the application as an ordinary library. Meanwhile, SRandomizer simultaneously performs the relocation of the stand-by module in the background. The location of the module is randomly determined in the 64 bytes of aligned memory address. After relocation, SRandomizer adjusts pointers of the stand-by module so that they contain valid memory addresses. When the module is ready to be executed, its role is changed to *activated*. SRandomizer repeatedly changes the role of those modules at a short fixed interval (i.e., 50 ms), which can be achieved by modifying the corresponding PLT (i.e., .plt in Figure 2) with synchronization. By doing so, a target application gets shielded by SRandomizer without any interruption, and without even being notified of the randomization process. Synchronization is needed only when the PLT table is updated, so it does not block the execution of the threads for a long time.

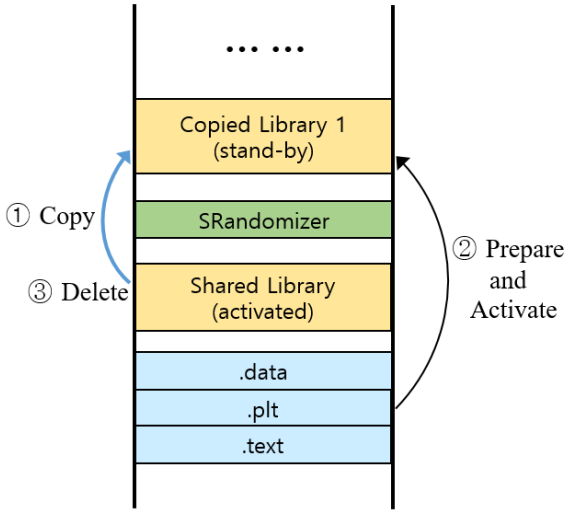

Figure 2. SRandomizer approach for rerandomizing a shared library.

Our technique leverages the ELF binary structure, which allows an executable with a position independent code (PIC) [26] to reside at any memory location without a source code recompilation. We achieved the transformation by implementing a runtime relocating module in SRandomizer. This module performs repeated memory relocation of a normal PIC module in the process. In order to inject SRandomizer into a running process, we utilized ptrace, which is a user-level subsystem for process control in Linux. ptrace does not attach a debugger onto a process; thus, using this is much more time efficient than restarting the process. This is especially relevant when a heavy process has a database or a network connection.

Algorithm 1 shows the pseudo-code of SRandomizer. First, it initializes the data structures, such as PLT information, and sets base addresses and offsets of the shared libraries. Second, it generates random address space with the granularity of 64 bytes, which will be a new starting address of the shared library within some area in the *Probe_location()* function. After it successfully gets a candidate destination address, it locks a spin lock to ensure mutual exclusion. *Memory_copy()* allocates memory by using a *mmap()* call, copies a text segment, and updates the PLT table. *Adjust_pointers()* adjusts the pointers in the data segment. After memory copying and adjusting the pointers, SRandomizer unlocks the spin lock. Finally, it activates the new copied library. In Algorithm 1, SRandomizer randomizes one shared library. Although SRandomizer can randomize multiple shared libraries, it randomizes one critical shared library, such as *libc.so*, because the ROP attack in the real world must use this library, which includes the *system()* function.

Although other solutions [11,24] must perform pointer tracking, SRandomizer only needs to update the PLT table. This makes SRandomizer simple and efficient, and ensures SRandomizer

does not have a large overhead, as described in [27]. We describe the correctness and efficiency of SRandomizer in the evaluation section.

Algorithm 1 SRandomizer.

```
1:  procedure THREAD_MAIN()
2:      Initialization
3:      while !abort-signal do
4:          src = Restore_location(plts)
5:          dest = Probe_location(area, size)
6:          if (src!=-1 and dest!=-1) then
7:              if (!pthread_spin_lock(lock)) then                    ▷ locking
8:                  if (Memory_copy(dest, src, size)) then
9:                      Adjust_pointers(data_seg_addr, data_seg_size, cur_base_addr)
10:                     pthread_spin_unlock(lock)                     ▷ unlocking (1)
11:                     Activation(dest)
12:                     Release_region(src, size)
13:                     Usleep(interval)
14:                 else
15:                     pthread_spin_unlock(lock)                     ▷ unlocking (2)
16:                     Release_region(dest, size)
17:                 end if
18:             else
19:                 Release_region(dest, size)
20:             end if
21:         end if
22:     end while
23: end procedure
```

Figure 3 shows how to use SRandomizer. As you see in this figure, SRandomizer can be used with the target process without interrupting it. This is the reason why it is implemented as a shared library (i.e., SRandomizer.so), which can be easily injected into a running application (e.g., libvirtd), as shown in the figure. In the current work, we limit the scope of the target modules which SRandomizer can randomize to an important shared library linked in the process. Such a strategy is based on the fact that most code-reuse attacks target widely-used standard shared libraries, such as *libc.so*, since they use the required components for the attack. The C standard library *libc.so* provides macros, type definitions, and functions for tasks such as string handling, input/output processing, memory management, and several other operating system services. The code-reuse attack has to use the C standard library because it wants to engage in malicious behavior after the attack whilst ensuring conciseness and effectiveness. Although we randomize the shared library *libc.so* in this paper, it is possible to randomize any shared library file for protection from the code-reuse attack.

```
user@ubuntu:~/05_SRandomizer/SRand$ sudo ./inject -n libvirtd SRandomizer.so
targeting process "libvirtd" with pid 7784
"SRandomizer.so" successfully injected
user@ubuntu:~/05_SRandomizer/SRand$ _
```

Figure 3. Execution result.

4. Evaluation

In order to evaluate the correctness, security, and performance of SRandomizer, we considered four criteria:

(1) The correctness of adjusting pointers;
(2) A theoretical address space analysis;

(3) A real ROP exploit test;
(4) A performance comparison of SPEC CPU benchmarks.

4.1. Experiment Setup

In order to evaluate SRandomizer, we conducted some experiments on a machine with a 3.0 GHz Intel(R) Core i7 CPU and 16 GB RAM running a 64-bit Ubuntu Linux 16. For the experiment, we used a real application—Linux virtualization server daemon—that serves various virtual machines to the virtualization clients in Figure 4. The client apps request services to the *libvirtd* daemon by using a request URI. The libvirt driver daemon, *libvirtd*, is the basic building block for libvirt functionality to support the capability to handle specific hypervisor driver calls [28]. Finally, the libvirt API provides a virtualization service to the client. Two functions are important in the *libvirtd* process in order to defend the ROP attack: the *system()* function to execute the */bin/sh* shell, and the *setresuid()* to set the effective user ID. These functions are in the C standard library and essential to get a root shell in the Linux system. Hereafter, we show how SRandomizer is effective by measuring the data of these important functions. After all, we aim to evaluate that a virtualization daemon equipped with SRandomizer is protected well against the ROP attack. This virtualization daemon (i.e., *libvirtd*) is a critical application in cloud computing.

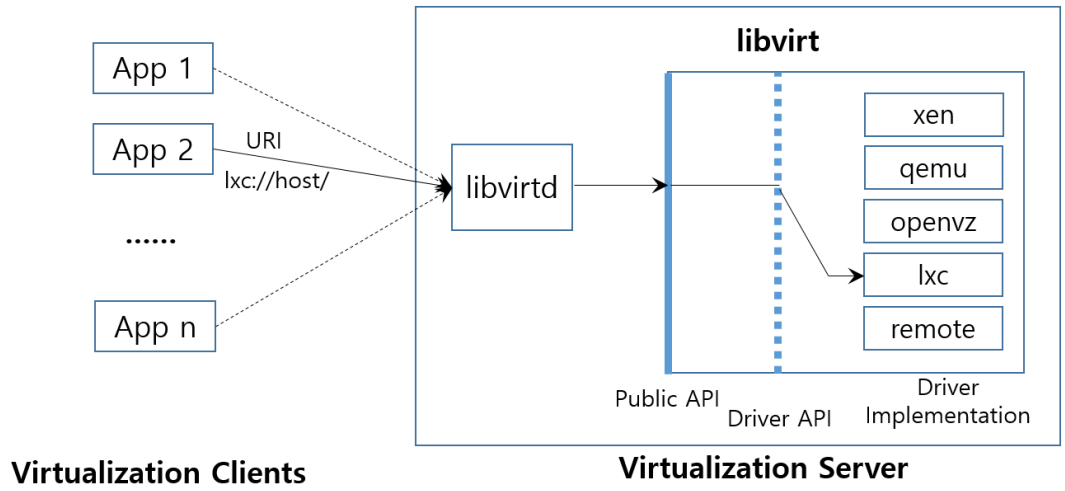

Figure 4. Libvirtd workflow (a case study for cloud computing).

4.2. Correctness

We first evaluated the correctness of the pointer identification in SRandomizer. Because SRandomizer modifies only pointers in the data segment of a shared library and the pointers' value is usually greater than 0x7fc200000000, it is almost impossible to confuse an integer and a pointer. For a data segment of size s bytes in a 64-bit address space, the probability p of an integer pointing to the data segment region is $p = s \times \frac{1}{2^{64}}$. Usually, the size of a data segment is below 1 MB. For example, the size of a *libc.so* file is 1.8 MB and the data segment is smaller than 1 MB. Therefore, the probability of confusing them is $p \approx 2^{20} \times \frac{1}{2^{64}} = 2^{-44}$, which is negligible.

In addition, we evaluated the false positive and false negative rates of the adjusted pointers and checked whether the memory address was rerandomized correctly. The goal of false pointer verification was to scour the data segment for 8-byte pointers and verify that our adjustment found all (no false negatives) pointers and not too many (no false positives) pointers. Verification was performed right before and after rerandomization, when it is important that every pointer has been adjusted correctly. We ran the libvirtd daemon with SRandomizer for the analysis, and experiments were repeated 100 times. The results show a total of 751 pointers were moved on average with no false positives; i.e., all moved pointers indeed pointed into the valid memory region. In addition, no false negatives were detected.

4.3. Security

(1) Formal Analysis

SRandomizer carries out module-level rerandomization on the shared library. SRandomizer equips mremap with a secure pseudorandom number generator (i.e., /dev/random). In SRandomizer, we generated the random base address as the form 0x7fc2???????0, where ? bits were randomized. This means that we configured the entropy of the memory layout as 28 bits, which is Linux's default setting. To evaluate the security, we ran *libvirtd* with SRandomizer. We verified that the question-marked bits in 0x7fc2???????0 were continuously randomized in actuality. Although an attacker comes to know the memory layout at T_i, it is useless at $T_j (j > i)$ because of rerandomization. Therefore, the attacker's ordeal becomes a random game. For n-bit random addresses, the probability that a random guessing attack succeeds at T_j is $p = \frac{1}{2^n}$. Because we used the 28-bit randomness, the probability of a successful attack is $\frac{1}{2^{28}}$. This means that the expected number of probes required is 2^{28} = 268,435,456.

Figure 5 shows the variation example of the address of the system function in the *libvirtd* daemon (i.e., system()) over time. We can see the high entropy of distributed addresses in this figure, which means that the probability of a random address occurring is almost a uniform distribution. In addition, SRandomizer provides a randomization with a granularity of 64 bytes, which is more fine-grained than the ASLR solutions in common OSs providing page-level (i.e., 4 KB) randomization. On the basis of these results, we conclude that it is impossible for an attacker to guess the location of the function in time.

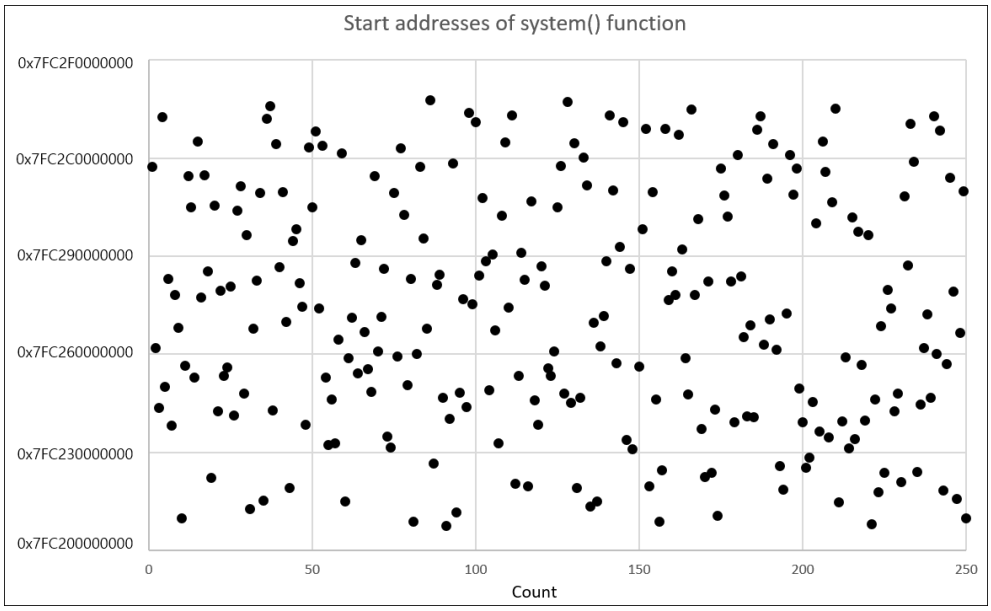

Figure 5. One example of system() start addresses over time (count).

(2) Real Exploit Testing in Cloud Computing Platform

To ensure the effectiveness of SRandomizer, we also tested whether SRandomizer could defend against a real ROP attack. We used a just-in-time ROP (JIT-ROP) exploit [5] vulnerable libvirtd 1.3.1. The exploitation consisted of several steps: initializing memory disclosure, automatically harvesting code pages for the exploit, finding necessary functions and gadgets, and just-in-time compiling the attacker's program. To check the effectiveness of SRandomizer, we ran *libvirtd* with and without SRandomizer. The results are shown in Figures 6 and 7. In the case of *libvirtd* without SRandomizer, JIT-ROP successfully bypassed ASLR on the first try. However, when SRandomizer was enabled, the JIT-ROP attack failed in all of the 100 attempts. The reason why the ROP attack cannot work under

SRandomizer is clear: the guessed address becomes stale in later attempts because of rerandomization. As a consequence, we conclude that SRandomizer can protect critical applications such as hypervisors of cloud computing from disruptive influences. Furthermore, it is helpful for strengthening cloud systems, since hypervisors, which are usually an attacker's favorite target, play a key role in cloud systems.

Furthermore, SRandomizer follows the approach of a runtime binary randomization by repeating the binary relocation at extremely short intervals (i.e., 50 ms) in a randomized manner. According to Shuffler [11], a 50 ms shuffle period is enough to prevent existing JIT-ROP attacks, which take 2.3 to 378 s to complete. In our experiments, JIT-ROP attacks took at least 4.6 s. Hence, any kind of attack, including ROP, and even advanced JIT-ROP techniques, will be denied unless the construction of an exploit code is completed within the time limit, since knowledge of the address of an exploitable code at time t_i becomes useless at time $t_j (j > i)$.

As mentioned in Section 2.2, there are several rerandomization techniques against JIT-ROP attacks. Shuffler [11] performs rerandomization at a time interval of 50 ms, but it requires compiling, linking, re-execution, and binary rewriting. TASR [10] requires kernel and compiler modifications. RuntimeASLR [24] also needs a heavyweight instrumentation for rerandomization of a child process. However, SRandomizer provides efficient and effective rerandomization without these superfluous tasks, because it only randomizes shared libraries essential for gadget harvesting.

```
user@ubuntu:~/05_SRandomizer/ROP$ python ROP_exploit.py
==========================================
Now, Let's start ROP attack.
Searching printf offset - Found!
Searching system offset - Found!
Searching setresuid offset - Found!
Searching binsh string - Found!
Searching gadgets - Found!
Sending the payload.
[*] Switching to interactive mode
$ id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),110(lxd),115(lpadmin),116(sambashare),117(kvm),121(libvirtd)
$
```

Figure 6. ROP attack against libvirtd without SRandomizer.

```
user@ubuntu:~/05_SRandomizer/ROP$ python ROP_exploit.py
==========================================
Now, Let's start ROP attack.
Searching printf offset - Found!
Searching system offset - Found!
Searching setresuid offset - Found!
Searching binsh string - Found!
Searching gadgets - Found!
Sending the payload.
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ id
[*] Process 'libvirtd' stopped with exit code -11 (SIGSEGV) (pid 7784)
[*] Got EOF while sending in interactive
user@ubuntu:~/05_SRandomizer/ROP$
```

Figure 7. ROP attack against libvirtd with SRandomizer.

4.4. Performance

We evaluated the performance overhead of SRandomizer during the rerandomization. In our experiment, the execution times of the system() and setresuid() functions were measured with and without SRandomizer. As shown in Figure 8, the average execution time of system() without SRandomizer was 0.149 milliseconds, and it was 0.157 milliseconds for the case with SRandomizer, resulting in an around 5% performance drop. Figure 9 shows the execution time of setresuid(), indicating a 3% performance drop for SRandomizer. In the case where SRandomizer was disabled, the average execution time of the setresuid() function was 0.129 milliseconds. Since the performance

drop is less than 5% and the time difference is only 0.008 milliseconds, which humans cannot perceive, the overhead is negligible.

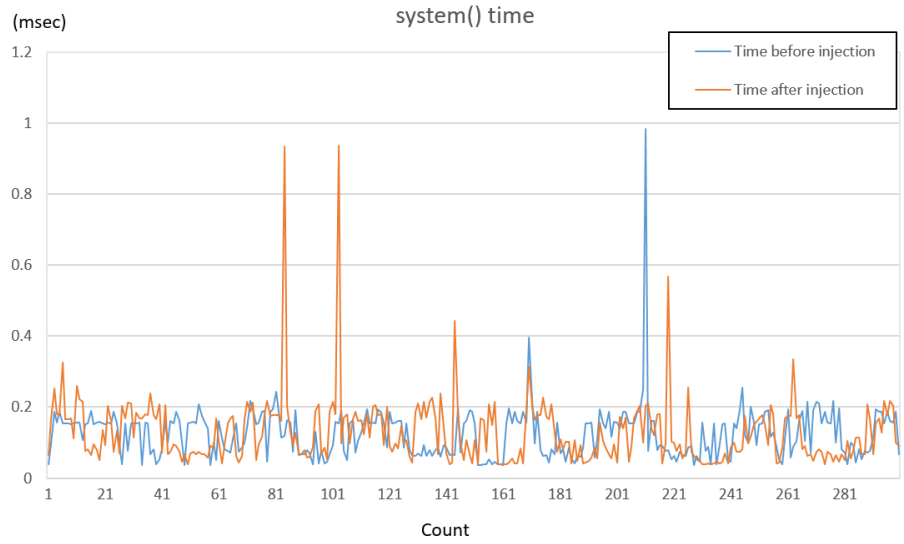

Figure 8. Performance overhead for system().

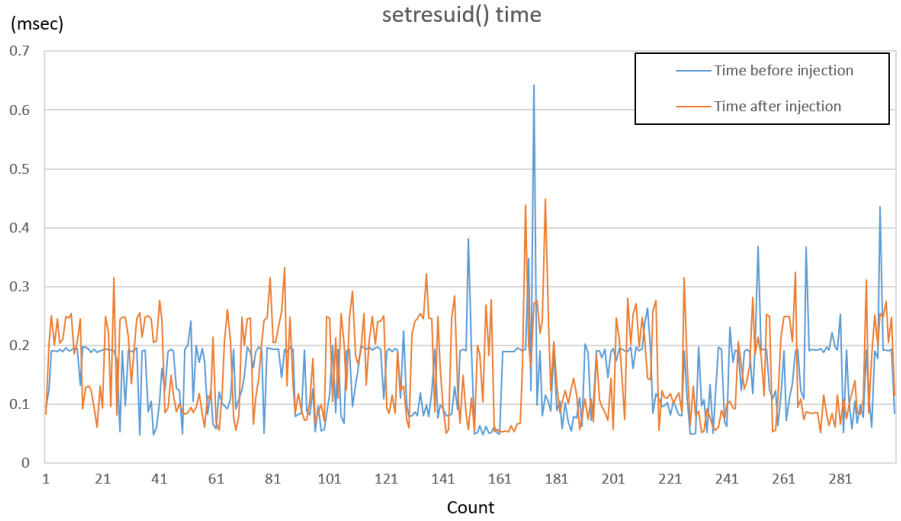

Figure 9. Performance overhead for setresuid().

We then evaluated SRandomizer with SPEC CPU2006 benchmarks, which consist of some real-world programs. Usually, SPEC CPU2006 benchmarks are used to test both the CPU and memory overhead because they are computationally-intensive benchmarks. Therefore, they have a limitation in the evaluation of rerandomization overheads for the programs using shared libraries. We compiled all the benchmark programs with options -pie -fPIC -O2. Figure 10 shows the results. The runtime overhead is only 2.1% on average and 4.2% in the worst case scenario. The low overhead associated with SRandomizer is due to the fact that it randomizes only the critical shared library; e.g., the C standard library.

Table 3 shows a comparison of the SPEC CPU2006 benchmarks' overhead. TASR [10] has a 2.1% average overhead, but they used the -Og debugging option in their evaluation, which makes the optimization 30% slower than the normal optimization level according to Shuffler [11]. Shuffler has a 14.9% average overhead, and RuntimeASLR [24] has a 1,221,838% average overhead. According to RuntimeASLR [24], this is due to the runtime taint analysis which is only performed at startup.

SRandomizer has an average overhead of only 2.1%. As a result, SRandomizer is more lightweight and efficient than TASR, Shuffler, and RuntimeASLR.

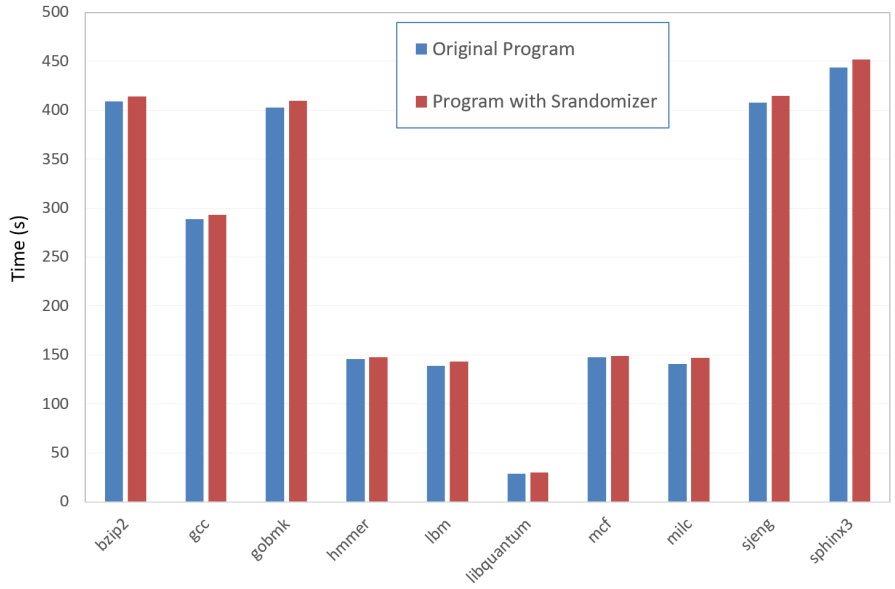

Figure 10. Runtime overhead for SRandomizer rerandomization on SPEC CPU2006 benchmarks.

Table 3. Comparison of SPEC CPU2006 benchmarks overhead.

	TASR	Shuffler	RuntimeASLR	SRandomizer
Average Overhead (%)	2.1% (debug option)	14.9%	1,221,838% (at startup)	2.1%

5. Discussion

SRandomizer is based on the position independent code (PIC); we discuss this problem in this section. In addition, we discuss the problem of an ROP attack direct to the main program.

5.1. ROP Attack Directly to the Main Program

SRandomizer is a form of shared library, and it rerandomizes important shared libraries; e.g., the C standard library. As a result of this, it can provide a seamless application service without interruption. We investigated CVEs from the past three years because we want to know that it is enough to rerandomize only shared libraries. After investigating, we concluded that, to our knowledge, it is enough to rerandomize shared libraries. All memory corruption vulnerabilities, such as buffer overflow and format string, have a real bug in the C standard library, as shown in Section 3.1. In addition, although there is a vulnerability in the main program rather than the shared libraries, SRandomizer is still useful because it rerandomizes the necessary shared libraries to obtain a shell.

5.2. Position Independent Code

We assumed the position independent code (PIC), which is a machine code that, being placed somewhere in the memory region, executes appropriately regardless of its absolute address space [26]. PIC is extensively used for shared libraries, so that the same library code can be loaded in the location of each program address space, and it will not overlap with any other memory uses, such as other shared libraries [29]. Unix-based operating systems provide a PIC environment, but the Windows operating system does not. Therefore, SRandomizer works on Linux, Unix, Mac OS, etc., but not on Windows. In Windows, the kernel memory manager manages ASLR. If Windows were to support GOT and PIC, SRandomizer would work on it.

6. Conclusions and Future Work

A seamless service is important, especially for mission-critical applications such as a hypervisor of cloud computing. However, previous ASLR solutions for seamless service have limitations to them: the need for system modifications, recompiling, and process restarting. In this paper, we proposed a novel ASLR technique in the form of a shared library. It is injected into a critical application without a process restart and then randomizes the shared libraries seamlessly. We described its design and implementation. Furthermore, we showed the execution results in a cloud system, regarding effectiveness and efficiency, through comparisons with related work. We expect this technique to be very helpful in defending against memory corruption attacks, especially for cloud systems. In the future, we will study randomization methods inside of a shared library and further elaborate our method.

Author Contributions: J.Y. and Y.S. contributed the ideas and wrote the paper; D.K. designed and conducted the experiments; K.-W.P. performed the security analysis. J.Y. supervised the whole paper, including paper organization and proofreading. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2019-2018-0-01423) supervised by the IITP (Institute for Information and communications Technology Promotion). Moreover, this research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (number NRF-2018R1D1A1B07047323) and by the Korean government (MSIT) (number NRF-2017R1C1B5077026). This work was supported by the Institute of Information and communications Technology Planning and Evaluation (IITP) grant funded by the Korean government (MSIT) (number 2018-0-00420 and No.2019-0-00533)

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dowd, M.; McDonald, J.; Schuh, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*; Pearson Education: Hoboken, NJ, USA, 2006.
2. PaX Team. Address Space Layout Randomization (ASLR). Available online: http://pax.grsecurity.net/docs/aslr.txt (accessed on 7 January 2019).
3. Microsoft. Data Execution Prevention (DEP). Available online: http://support.microsoft.com/kb/875352/EN-US/ (accessed on 7 January 2019).
4. Shacham, H.; Page, M.; Pfaff, B.; Goh, E.; Modadugu, N.; Boneh, D. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM conference on Computer and Communications Security (CCS 2004), Washington, DC, USA, 25–29 October 2004; pp. 298–307. [CrossRef]
5. Snow, K.Z.; Monrose, F.; Davi, L.; Dmitrienko, A.; Liebchen, C.; Sadeghi, A.-R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 574–588.
6. Bittau, A.; Belay, A.; Mashtizadeh, A.; Mazières, D.; Boneh, D. Hacking Blind. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014; pp. 227–242.
7. Satish, A.; Shiou, T.; Zhang, C.; Elmeleegy, K.; Zwaenepoel, W. Scrub: online troubleshooting for large mission-critical applications. In Proceedings of the Thirteenth EuroSys Conference (EuroSys '18), Porto, Portugal, 23–26 April 2018; pp. 1–15. [CrossRef]
8. Roemer, R.; Buchanan, E.; Shacham, H.; Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2012**, *15*, 1–34. [CrossRef]
9. Gupta, A.; Kerr, S.; Kirkpatrick, M.S.; Bertino, E. Marlin: A fine grained randomization approach to defend against ROP attacks. In Proceedings of the International Conference on Network and System Security (NSS 2013), Madrid, Spain, 3–4 June 2013; pp. 293–306.
10. Bigelow, D.; Hobson, T.; Rudd, R.; Streilein, W.; Okhravi, H. Timely Rerandomization for Mitigating Memory Disclosures. In Proceedings of the 22nd ACM conference on Computer and Communications Security (CCS '15), Denver, CO, USA, 12–16 October 2015; pp. 268–279.

11. Williams-King, D.; Gobieski, G.; Williams-King, K.; Blake, J.P.; Yuan, X.; Colp, P.; Zheng, M.; Kemerlis, V.P.; Yang, J.; Aiello, W. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), Savannah, GA, USA, 2–4 November 2016; pp. 367–382.
12. Backes, M.; Nürnberger, S. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 433–447.
13. Solar Designer. LPR LIBC RETURN Exploit. Available online: http://insecure.org/sploits/linux.libc.return.lpr.sploit.html (accessed on 10 January 2019).
14. PHRACK. The Advanced Return-Into-Lib(c) Exploits: PaX Case Study. *Phrack Mag.* **2001**, *58*. Available online: http://phrack.org/issues/58/4.html#article (accessed on 7 January 2019).
15. 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 '07), Alexandria, VA, USA, 29 October–2 November 2007; pp. 552–561.
16. Abadi, M.; Budiu, M.; Erlingsson, Ú.; Ligatti, J. Control-flow integrity. In Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05), Alexandria, VA, USA, 7–11 November 2005; pp. 340–353. [CrossRef]
17. Abadi, M.; Budiu, M.; Erlingsson, Ú.; Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2009**, *13*, 1–40. [CrossRef]
18. Frantzen, M.; Shuey, M. StackGhost: Hardware facilitated stack protection. In Proceedings of the 10th USENIX Security Symposium, Washington, DC, USA, 13–17 August 2001; pp. 55–66.
19. Dang, T.H.Y.; Maniatis, P.; Wagner, D. The Performance Cost of Shadow Stacks and Stack Canaries. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2015), Singapore, 14–17 April 2015; pp. 555–566.
20. Davi, L.; Liebchen, C.; Sadeghi, A.-R.; Snow, K.Z.; Monrose, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In Proceedings of the 22nd Annual Network and Distributed System Security Symposium NDSS, San Diego, CA, USA, 8–11 February 2015; pp. 1–15.
21. Crane, S.; Liebchen, C.; Homescu, A.; Davi, L.; Larsen, P.; Sadeghi, A.-R.; Franz, M. Readactor: Practical code randomization resilient to memory disclosure. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 763–780.
22. Backes, M.; Holz, T.; Kollenda, B.; Koppe, P.; Nurnberger, S.; Pewny, J. You can run but you can't read: Preventing disclosure exploits in executable code. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 1342–1353.
23. Friedman, S.E.; Musliner, D.J.; Keller, P.K. Chronomorphic programs: Runtime diversity prevents exploits and reconnaissance. *Int. J. Adv. Secur.* **2015**, *8*, 120–129. [CrossRef]
24. Lu, K.; Lee, W.; Nürnberger, S.; Backes, M. How to make ASLR win the clone wars: Runtime re-randomization. In Proceedings of the 23rd Annual Network and Distributed System Security Symposium NDSS, San Diego, CA, USA, 21–24 February 2016; pp. 1–15.
25. CVE-2018-6764. Vulnerability Details. Available online: https://www.cvedetails.com/cve/CVE-2018-6764/ (accessed on 10 October 2019).
26. Levine, J.R. *Linkers and Loaders*; Morgan-Kauffman: Burlington, MA, USA, 1999; pp. 170–171, ISBN 1-55860-496-0.
27. Shen, Z.; Chen, W. A Survey of Research on Runtime Rerandomization Under Memory Disclosure. *IEEE Access* **2019**, *7*, 105432–105440. [CrossRef]
28. Redhat Linux Project. The Libvirt API Concepts. Available online: https://libvirt.org/api.html (accessed on 10 December 2019).
29. Drepper, U. How to Write Shared Libraries. Available online: https://www.ukuug.org/events/linux2002/papers/pdf/dsohowto.pdf (accessed on 10 December 2019).