Derandomizing Kernel Address Space Layout for Memory Introspection and Forensics

Yufei Gu
The University of Texas at Dallas
800 W. Campbell RD
Richardson, TX 75080
yufei.gu@utdallas.edu

Zhiqiang Lin
The University of Texas at Dallas
800 W. Campbell RD
Richardson, TX 75080
zhiqiang.lin@utdallas.edu

ABSTRACT

Modern OS kernels including Windows, Linux, and Mac OS all have adopted kernel Address Space Layout Randomization (ASLR), which shifts the base address of kernel code and data into different locations in different runs. Consequently, when performing introspection or forensic analysis of kernel memory, we cannot use any pre-determined addresses to interpret the kernel events. Instead, we must derandomize the address space layout and use the new addresses. However, few efforts have been made to derandomize the kernel address space and yet there are many questions left such as which approach is more efficient and robust. Therefore, we present the first systematic study of how to derandomize a kernel when given a memory snapshot of a running kernel instance. Unlike the derandomization approaches used in traditional memory exploits in which only remote access is available, with introspection and forensics applications, we can use all the information available in kernel memory to generate signatures and derandomize the ASLR. In other words, there exists a large volume of solutions for this problem. As such, in this paper we examine a number of typical approaches to generate strong signatures from both kernel code and data based on the insight of how kernel code and data is updated, and compare them from efficiency (in terms of simplicity, speed etc.) and robustness (e.g., whether the approach is hard to be evaded or forged) perspective. In particular, we have designed four approaches including brute-force code scanning, patched code signature generation, unpatched code signature generation, and read-only pointer based approach, according to the intrinsic behavior of kernel code and data with respect to kernel ASLR. We have gained encouraging results for each of these approaches and the corresponding experimental results are reported in this paper.

CCS Concepts

•Security and privacy → Operating systems security; Operating systems security; Virtualization and security; •Applied computing → System forensics;

CODASPY'16, March 09 - 11, 2016, New Orleans, LA, USA

ISBN 978-1-4503-3935-3/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2857705.2857707

Keywords

Kernel Address Space Layout Randomization; Virtual Machine Introspection; Memory Forensics

1. INTRODUCTION

Address space layout randomization (ASLR) [27] has become a prominent defense against the attacks that use a hard-coded address to compromise vulnerable systems. Examples of such attacks include Internet worms that use the same virtual address to compromise the control flow of the same vulnerable program, or some kernel rootkits that overwrite the same virtual address to hide or redirect the kernel control flow. At a high level, ASLR randomizes the base address of program code and data including both heap and stack. Consequently, traditional memory exploits through return-into-libc [10, 19] or return oriented programming (ROP) [24, 9] can be mitigated because of the memory address diversity enabled by ASLR. ASLR has also been pushed to the kernel space due to the existence of the exploitable vulnerabilities in OS kernels as well as the threats from kernel rootkits. Modern OS kernels such as Windows, Linux, and Mac OS all have adopted ASLR to randomize both the kernel code and the kernel data including those in kernel global, heap and stack regions. As such, the address of kernel code and data (e.g., system call dispatcher table) will be relocated to different memory locations in different runs.

The implication of kernel ASLR has twofold: on one hand it significantly decreases the success rate of kernel memory exploits as well as some kernel rootkit attacks; on the other hand it also hinders the application of online kernel introspection [14] and offline kernel memory forensics, both of which need to interpret (or reconstruct) kernel events outside of the (guest) OS. Specifically, for an introspection and forensic tool to be effective, it often requires a pre-knowledge of the OS kernel such as where kernel code and important kernel data structure is located. For instance, to interpret a system call event, it requires to know the address of the system call tables (e.g., [13]); to intercept the kernel object allocation and deallocation, it requires to know the addresses of the functions that manages the kernel heaps (e.g., [30]); to traverse certain dynamically allocated kernel objects, it needs to know their rooted global addresses (e.g., [12]). Unfortunately, kernel ASLR will randomize these addresses, and we must derandomize them for introspection and forensics.

From the offense perspective, there are already several attempts to derandomize the user space ASLR. In particular, Shacham et al. [25] demonstrated the first brute-force linear search approach, which only requires 2^{16} probes at worst (2^{15} on average) to derandomize the address space of a vulnerable program for a 32-bit ASLR implementation. Such a brute-force approach was also used in recent BROP [8] attack to bypass the ASLR protection. Additionally, another way

to derandomize ASLR is through information leakage. Roglia et al. [21] demonstrated a surgical approach to return to randomized `libc` by exploiting information about the base address of `libc` in victim process memory and also combing the code fragments available at fixed locations and use them to discover the address of other `libc` functions. Meanwhile, unlike this single memory disclosure, recent JIT-ROP [26] attack leverages multiple memory disclosures to bypass fine-grained ASLR through repeatedly abusing a memory disclosure to map an application's memory layout on-the-fly and dynamically discover the attack gadgets.

Interestingly, while we can use these offensive techniques, which have only the *remote access* to derandomize the user space ASLR, we have the *local access* of the entire memory for introspection or forensics applications and we can leverage such an advantage to derandomize the kernel ASLR. For instance, memory forensics tools such as Volatility [29] uses a KDBG signature (a sequence of bytes) to derandomize Windows kernel address space. In other words, there are too many options (e.g., too many signatures) to perform the derandomization when having the physical access of the kernel memory. Given such a large volume of solution space, there is however no study that has searched for the optimal solutions in terms of both robustness (i.e., hard to evade) and efficiency (i.e., having a fast performance).

Therefore, in this paper, we conduct the first systematic study to search for the optimal solutions for introspection and forensics to derandomize the kernel ASLR. In particular, since the key challenge lies in deriving the strong and robust signatures inside kernel memory, we systematically examine both kernel read-only code and data that can be used to derandomize the ASLR. For read-only kernel data, we examine the strings and entries of code pointers (e.g., jump tables) and we propose to use the entries of the code pointers as the signatures. For kernel code, we examine how kernel code is updated, from which to derive the robust signatures. We also perform a comparison study among these approaches by using robustness and efficiency metrics.

In summary, this paper makes the following contributions:

- We make the first systematic study in searching for the optimal solutions of derandomizing kernel ASLR for virtual machine introspection and memory forensics.
- We revisit, examine, and devise four different approaches from kernel code and data perspective. Among them, three are novel approaches that have not been reported, and they explore the intrinsic properties of kernel code and also the way of how kernel code and data is updated.
- We have implemented these approaches and compared them using the metrics of robustness and efficiency. We have tested 20 recent Linux kernels, and the detailed experimental results are reported in this paper.

2. BACKGROUND AND RELATED WORK

Software is so complicated today especially for an OS kernel, and it contains inevitable vulnerabilities. As discussed earlier, due to the existence of exploitable vulnerabilities inside OS kernels and those rootkit attacks, modern OSes all have pushed ASLR into kernel space, for instance:

- **Microsoft Windows**. Starting from Windows Vista (released in January 2007) [2], Microsoft has enabled ASLR inside the kernel space.
- **Linux**. Starting from the kernel version 3.14 (released in March 2014) [5], Linux supports the kernel ASLR. While currently it is disabled by default, users can turn it on by configuring the kernel compilation options and then rebuilding the kernel.
- **Mac OS**. Starting from OS X Mountain Lion 10.8 (released in July 2012) [4], the entire system (include the kernel) supports ASLR.

Therefore, kernel ASLR has become a de facto standard for modern OS. At a high level, it relocates both kernel code and data into different locations in different runs of the kernel. Consequently, we must derandomize the kernel address space layout before performing the introspection or forensics analysis.

State-of-the-art. A straightforward approach to derandomize the kernel ASLR would be devising strong signatures from OS kernel code or data, and then searching for them to derandomize the address. Volatility [29], a memory forensic analysis tool, uses such an approach to derandomize the Windows kernel (currently it does not support Linux kernel yet). Specifically, Volatility uses the Windows `KdDebuggerDataBlock` (KDBG), a data structure maintained by the Windows kernel for debugging purposes. KDBG contains a list of the running processes and loaded kernel modules. It also contains some version information. Identifying this data structure in memory can reveal many useful information including certain code addresses of the kernel. Meanwhile, the header of this data structure also contains some unique binary for different Windows versions, which can serve as signatures.

Therefore, Volatility contains a `kdbgscan` plugin, which is particularly designed to scan the KDBG, from which to derandomize the ASLR. A list of the signatures used by `kdbgscan` is presented in Table 1. We can notice that these signatures are very short (at most 14 bytes in length), and more importantly, these data are located in writable kernel data sections, which means a non-collaborative guest OS can easily cheat the introspection or the forensics tool. We have verified that the signatures used by `kdbgscan` can be easily modified without crashing the kernel. Therefore, we need robust signatures to derandomize the kernel ASLR. Our paper focuses on how to derive such robust signatures.

Other related works. There are also efforts focusing on fingerprinting guest OS kernel version. While derandomzing kernel ASLR and kernel version fingerprinting are different problems, they share certain similarity in that they both have to derive and search for strong signatures from an OS kernel. Meanwhile, kernel version fingerprinting can be used as a first step for derandomizing the ASLR for a specific kernel, though it might also be possible to use the strong and unique signatures to directly derandomize the kernel ASLR without the fingerprinting step. Therefore, kernel fingerprinting technique can help derandomize the kernel ASLR. The signatures used in the kernel ASLR derandomization can also complement the kernel fingerprinting depending on whether they are unique or not.

In the past a few years, there are a number of efforts focusing on how to fingerprint the guest OS version when having the physical access of the computer (e.g., in the cloud environment for cloud providers). UFO [20] is one such a system that explores the discrepancies in the CPU state for different OSes. By profiling, extracting, and differing the values in CPU registers such as GDT, IDT, CS, CR, and TR, UFO can generate unique signatures for a family of Windows kernels.

OS-Sommelier [15] is another system that explores robust signatures from kernel code to fingerprint the guest OS. More specifically,

Kernel Version	Signature (Byte Sequence)	Size (Bytes)
VistaSP0x86	00 00 00 00 00 00 00 00 4b 44 42 47 28 03	14
VistaSP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
VistaSP2x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
VistaSP0x64	00 f8 ff ff 4b 44 42 47 28 03	10
VistaSP1x64	00 f8 ff ff 4b 44 42 47 30 03	10
VistaSP2x64	00 f8 ff ff 4b 44 42 47 30 03	10
Win7SP1x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win7SP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 40 03	14
Win7SP0x86	00 00 00 00 00 00 00 00 4b 44 42 47 40 03	14
Win7SP0x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win2008SP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
Win2008SP2x86	00 00 00 00 00 00 00 00 4b 44 42 47 30 03	14
Win2008SP1x64	00 f8 ff ff 4b 44 42 47 30 03	10
Win2008SP2x64	00 f8 ff ff 4b 44 42 47 30 03	10
Win2008R2SP0x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win2008R2SP1x64	00 f8 ff ff 4b 44 42 47 40 03	10
Win8SP0x86	00 00 00 00 00 00 00 00 4b 44 42 47 60 03	14
Win8SP1x86	00 00 00 00 00 00 00 00 4b 44 42 47 60 03	14
Win8SP0x64	03 f8 ff ff 4b 44 42 47 60 03	10
Win8SP1x64	03 f8 ff ff 4b 44 42 47 60 03	10
Win2012x64	03 f8 ff ff 4b 44 42 47 60 03	10
Win2012R2x64	03 f8 ff ff 4b 44 42 47 60 03	10

Table 1: KDBG Signatures used by Volatility to Derandomize the Kernel.

it computes the core kernel code hash to precisely fingerprint an OS, and these core kernel code is identified by correlative disassembling, code and signature normalization, and resilient signature matching techniques. OS-Sommelier$^+$ [16] further combines kernel data structure for the fingerprinting. Instead of precisely identifying the core kernel code from the memory snapshot and computing the hash, Sdkernel [23] utilizes an approximate matching tool Sdhash [22] to extract kernel fingerprints from the content of the disk images. Sdkernel would work well for disk forensics but not on memory since kernel code can be significantly changed due to the dynamic patching issues discussed in §3.2.

Most recently, Ahmed et al. [6] proposed the use of relocation tables in the program binary code to compute their fingerprints. Their key idea is that relocation tables tend to be distinct, and the relative addresses among the relocation entries can hence be used to build unique signatures. Their experimental results show that this approach can achieve very high accuracy but not 100% for Windows binaries (they have not tested any Linux binaries yet).

3. OVERVIEW

In this section, we give an overview of what we aim to achieve in this paper. We first define our research problem in §3.1, then enumerate the challenges faced to derandomize the kernel ASLR in §3.2, and finally present an outline of the approaches we will study in §3.3.

3.1 Problem Statement

The goal of our work is to investigate the optimal solutions for derandomizing the kernel address space for introspection and forensics. Under such application scenarios, we have the physical access of the target computers, and consequently there exist a large number of solutions for this problem. Therefore, we would like to also answer the questions of what the solution space is and which solution is more optimal. To this end, we define two metrics to evaluation the possible solutions.

- **Robustness:** Since there could exist non-cooperative guest OS running in a cloud (e.g., criminals who want to defeat the memory forensics, or kernel rootkits that have tampered with the kernel memory), the signatures generated for the derandomization should be robust; namely, it should be quite challenging for an adversary to modify the signatures, or generate fake ones to mislead the derandomization process.

- **Efficiency:** Given the fact that the size of the kernel memory is usually very large and there could also be millions of VMs running in a cloud, we would prefer faster approaches — the faster the derandomization takes (or the simpler the approach is), the better.

Threat Model, Scope, and Assumptions. We focus on derandomizing the kernel ASLR for cloud providers or forensic investigators where they have the physical access of the OS memory. We assume there are non-cooperative cloud users (e.g., cyber criminals), or there exists kernel malware which can manipulate or forge the signatures. Also, we focus on x86 platform, and the OS we aim to derandomize are the recent Linux kernels since version 3.14. Meanwhile, we do not attempt to compare all the possible approaches, and instead we would like to design and compare the approaches that tend to be simple, robust, and efficient.

3.2 Challenges

Intuitively, while we can use those sophisticated offensive techniques such as the brute-force probing [25] or memory disclosure attack [21, 26] to derandomize the kernel ASLR, a more efficient approach would be to derive robust signatures from both kernel code and data, and use them for the derandomization. Therefore, the central problem we aim to solve is how to derive such signatures. In the following, we discuss the challenges faced during this step.

3.2.1 Kernel Code is Non Static

The most straightforward approach is to directly use the entire kernel code as the signatures, and search for the memory to locate them. However, such an approach cannot have 100% accuracy because kernel code is actually non static [17], and there exists various complicated kernel patching techniques during the kernel loading and even during the kernel run-time. More specifically, modern Linux kernel, the target of our work, often involves the following dynamic kernel code patching:

- **Relocation**. Relocation is typically needed by a linker when linking object code to produce the final executable for user space program. Relocation is also needed when loading kernel modules or loading ASLR-enabled kernel. Specifically, as the current ASLR basically shifts the base address of kernel code and kernel global data, there is a need to dynamically patch the static hard coded addresses in both kernel code (e.g., certain memory address operand) and kernel data (e.g., jump table entries in read-only global data sections). The location of these static addresses are described in the relocation entries in relocation table sections of the binary code (e.g., `.rel.text`, `.rel.data` and `.rel.rodata` sections). Two examples of the relocation patching are illustrated in Fig. 1. We can see from the first example that when this `mov` instruction gets loaded into different memory locations, its target memory address operand has been accordingly patched (e.g., from `0xa7b000` to `0xcb7b000`).

- **Alternative Instructions**. One optimization strategy used by modern Linux kernel is to dynamically replace some (old) instructions with more efficient alternatives. The benefits of this mechanism is to allow distributors to ship generic kernels which can then be self-optimized according to the

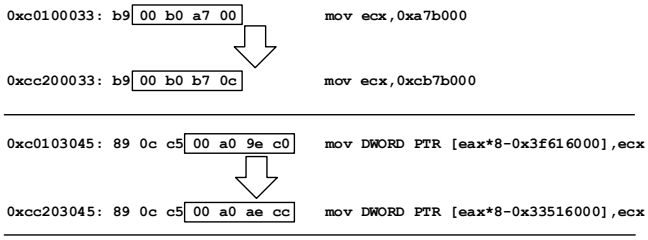

Figure 1: Relocation patching.

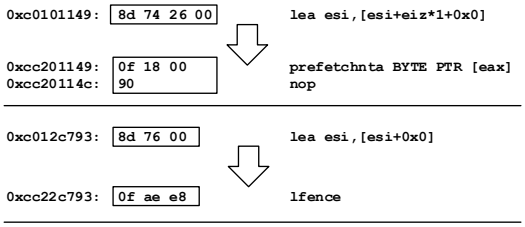

Figure 2: Alternative instruction patching.

CPU configuration at load time. For instance, the code built for older CPUs can take advantage of the alternative instructions added later in the newer CPUs. To use such patching, kernel developers have to explicitly declare the instruction substitution through macro definition statement in the kernel source code. For example, the following code snippet shows how an old `lock` and `addl` instruction sequence is replaced by `mfence` instruction if the CPU has XMM2 enabled:

```
#define mb() alternative("lock; addl $0,0(%%esp)",
"mfence", X86_FEATURE_XMM2)
```

These alternative patching definitions will be translated by compilers and then stored in special data sections such as `.altinstructions` and `.altinstr_replace` section in the kernel binary code. The kernel will apply alternative instructions by invoking `apply_alternative` function at load time. Fig. 2 also shows two examples of this alternative instruction patching.

- **Symmetric Multiprocessing**. In addition to the relocation and alternative instruction patching, kernel also has some other special patching. One example is the critical section locking and unlocking of the execution of Symmetric Multi-Processing (SMP) CPUs [3]. Note that SMP is an architecture that allows multiple CPUs to share the same memory. It is widely used in modern computers.

  Since in SMP mode, multiple CPUs can simultaneously access the same piece of memory, there are some regions of the kernel code that would become critical sections. In this situation, the critical section must be locked. However, kernel only activates these locks if it is operating in a multiple CPU environment. The SMP patching can occur at both load time and run time. During the loading phase, if kernel detects it runs in SMP mode, it will apply the SMP unlock and lock patching, as illustrated in Fig. 3.

  Additionally, the Linux kernel also supports enabling and disabling SMP CPUs at run time. This makes such patching

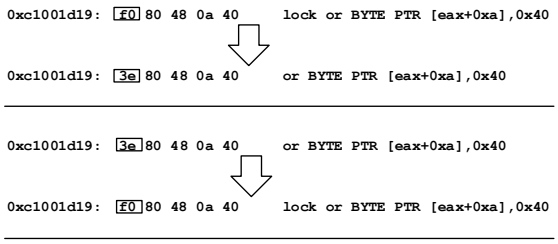

Figure 3: SMP unlock and lock patching.

occur at run time. In particular, Linux kernel uses the functions `alternatives_smp_lock` and `alternatives_smp_unlock` to a live kernel in memory to add or remove locks.

- **Function Tracing**: Function tracing is a mechanism which requires runtime code patching. The tracer is usually used to debug the kernel or measure performance. It is commonly called at the beginning of each function within the kernel. For performance reasons, each tracer call is replaced by a NOP slide when the tracing feature is currently disabled. For example, instruction `call mcount` is patched by a NOP instruction, e.g., `xchg %ax, %ax`. As such, the system will run with virtually no overhead when function tracing is disabled.

  Similar to relocation patching, function tracing patching is also informed by special data sections (e.g., `__mcount_loc`) that tracks where these tracing functions are located in the `.text` section. This special data section is generated during the compile time. Then during the kernel booting phase, before SMP is initialized, kernel will scan this special data section and update all the function tracing call site into NOPs. When tracing is enabled, the NOPs are patched back to calls.

In addition to those outlined above, there also exists other sophisticated kernel code patching such as jump label optimization and load-time hypercall patching [17]. They all show strong evidence that kernel code is non static and we have to deal with this challenge while deriving code signatures.

3.2.2 Kernel Data is Huge

Unlike kernel code (which tends to be small), there is a huge volume of kernel data located in different data sections (e.g., global, heap, and stack) in the memory. Apparently, writable data such as those in kernel heap and stack cannot be used as signatures, though their shape might be able to serve as the signatures [18, 28]. Therefore, the most intuitive approach would be to use the read-only data (such as the strings) as the signatures. However, string may be manipulated by adversaries [11, 7], and we must use the immutable ones. As a result, the key challenge lies in how to search for the unique and immutable kernel data and use them as the signatures.

3.3 Study Overview

Again, the goal of this work is to explore the possible optimal approaches to derandomizing kernel ASLR for introspection and forensics applications, and compare them in terms of robustness and efficiency. Since a program including OS kernel is composed of code and data, we divide the possible approaches into code-based and data-based, as illustrated in Fig. 4.

- **Kernel Code-based Approaches**. Modern Linux kernel contains several mega-bytes of code. Thus, too much information

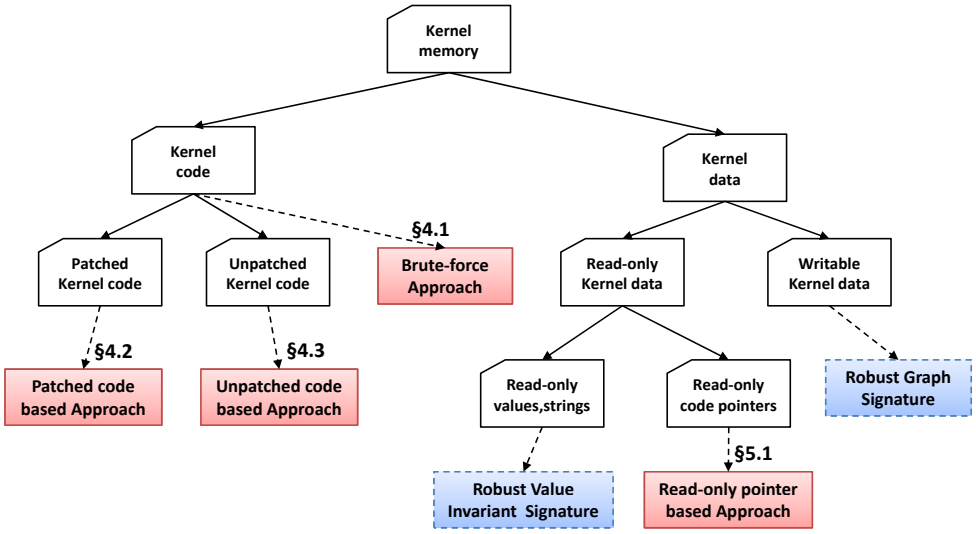

Figure 4: An Overview of the Investigated Approaches in Our Study.

can be used as the signatures. At a high level, the longer the signature is, the more robustness it is (because adversaries need to spend more efforts forging it for instance). On the other hand, we would prefer less sophisticated approaches because we also aim for efficiency. Similar to the brute-force approach used to break the ASLR [25], one of the simplest approach would be to directly scan the entire kernel memory and locate the base address of the kernel code, and we call this brute-force-approach (§4.1). However, this approach is not ideal because kernel code is not static and it would not achieve a 100% byte-by-byte matching. Therefore, more natural approaches would be to generate the derandomization signatures based on the patched code and unpatched code:

  - **Signatures from Patched Code**. While kernel patching is so complicated, there could be approaches that leverage the rules in the patching and derive the signatures. We can hence explore the intrinsic properties enforced by the patching and generate the signatures, and we will demonstrate one such an approach in §4.2.

  - **Signatures from Unpatched Code**. If we can distill the patched code, the rest will be the unpatched one, from which we can generate the signatures. Certainly the challenge will be how to identify those unpatched code, we will discuss a novel approach we devised in §4.3.

- **Kernel Data-based Approaches**. Kernel data can be divided into writable kernel data, and read-only data. Correspondingly, there exist different approaches based on this:

  - **Writable kernel data**. For writable data such as those located in kernel heap, we cannot directly use their values as the signatures, and instead we may use their shapes of the data structure as signatures, as demonstrated by earlier approach SigGraph [18]. In this study, we will not evaluate the feasibility of such an approach as it is too sophisticated, though it is possible to be used.

  - **Read-only kernel data**. Then we can look at the kernel read-only data. Constant strings are one of such candidates. However, strings can be manipulated by strong adversaries without crashing the kernel. While it is possible to systematically identify these non-manipulable strings (e.g., through fuzzing as in robust value invariant signatures [11]), we believe again this is too sophisticated, and instead we would like to focus on other non-manipulable data. We will discuss another novel approach we designed in §5.1.

4. DERIVING SIGNATURES FROM KERNEL CODE

4.1 Brute-force Code Matching Approach

The first and simplest approach is to directly scan the kernel code in memory using the entire code image from the disk. For a 32-bits Linux kernel, there are a maximum of 16 bits used in the randomization according to the default kernel configuration. Therefore, assume the kernel code has K bytes, the worst case complexity of this byte-by-byte brute-force scanning would take $2^{16}K$ number of comparisons. While there could be some optimizations, such as only using a subset of pages instead of all disk code for the scanning.

However, a caveat is that there will be no 100% byte-by-byte matching of the in-memory code and the in-disk code because of the dynamic patching of the kernel code (as discussed in §3.2). Fortunately, there will be only one peak match ratio, since it is very unlikely that there are two copies of the same kernel code in the memory. Therefore, we present a rigorous algorithm in Algorithm 1 for this peak-value based brute-force approach we used in our evaluation. Basically, it uses a byte-by-byte matching for each page (by calling `PageMatching`, which returns how many bytes get matched) and if more than half page of the data gets matched (line 8), then it compares with the next page. If all the code in disk has been checked, and the final match ratio has the highest peak value, then we output the based address of the kernel code (line 15).

The advantage of this brute-force approach is that it is the simplest (no sophisticated analysis required), and has very strong robustness since it uses the entire kernel code as the signatures. The disadvan-

Algorithm 1: A Brute-Force Based Code Matching Approach

Data: Kernel Page Size 4096;
Input: Kernel memory snapshot: M with M_p pages; Kernel code in disk D with D_p pages;
Result: The base address of the randomized kernel code

```
1  begin
2      peak ← 0;
3      R ← 0;
4      for i ∈ {0..M_p} do
5          matched ← 0;
6          j ← 0;
7          while j < D_p do
8              M[i] ← GetVirtualPageContent (M, i);
9              if (k ← PageMatching (M[i], D[j]) and
                   k > 2048) then
10                 j ← j + 1; matched ← matched + k ;
11             else
12                 j← 0; break;
13         if ((j == D_p) and (matched/D > peak)) then
14             peak ← matched/D; R ← GetVirtualAddr
                   (M, i − j);
15     return R;
```

Algorithm 2: Relocation Entry Based, Patched Code Signature Matching Approach.

Data: V_b: the base address of the kernel in the disk; n: the number of the randomization bits; $PhyKernAlign$: the kernel address alignment for the randomized kernel, which is usually at page level granularity.
Input: Kernel memory snapshot: M; Kernel code in disk D;
Result: The base address of the randomized kernel code

```
1  begin
2      for i ∈ {0..2^n} do
3          ProbBaseAddr← 0xc0000000 + i * PhyKernAlign;
4          matched ← false;
5          for each relocation offset O_j ∈ {rel.text} and with
           type R_386_32 do
6              if (M[ProbBaseAddr+ O_j - V_b] - D[O_j - V_b]) !=
               M[ProbBaseAddr+ O_{j+1} - V_b] - D[O_{j+1} - V_b])
               then
7                  matched ← false;
8                  break;
9              else
10                 matched ← true;
11         if (matched) then
12             return ProbBaseAddr;
13     return 0;
```

tage is it is very slow as shown in our experiment in §6 and also it may have false negatives as discussed in §7 when facing strong adversaries.

4.2 Patched Code Based Approach

While there are too many instructions that can be patched, we notice that there are still certain rules we can leverage. One is that kernel must know where to patch and how to patch. In particular, for relocation-based patching, kernel needs to use the information stored in relocation entries such as `.rel.text`, which means we can also leverage them to locate where to patch; for alternative instructions, kernel binary code also stores what those alternative instructions are and how to patch them (by invoking the internal kernel function `apply_alternative`). While we can re-execute the logic implemented in `apply_alternative` to locate the alternative instruction patched code, it is less complicated to locate the relocation-based patched code. For function tracing patching, we also need to parse the offset stored in the corresponding special data sections, which tends to be complicated. Therefore, we eventually decide to only look into how to use the relocation entries that can be directly acquired by tools such as `readelf` for our derandomization.

According to the standard ELF format specification [1], a relocation entry in a 32-bit ELF file is defined as a record consisting of two fields: the `offset` field and the `info` field, where the `offset` field gives the location at which the patching needs to be applied. Therefore, one insight we have is we can use the `offset` information stored in `.rel.text` for each relocation entry to probe and locate them in the kernel memory. The other insight is the data that needs to be patched are static memory address. Compared with static memory address that are in the disk file, all the patched memory address should be shifted by a constant value, which is the randomized offset we aim to get.

In fact, relocation entry had recently been used by CodeIndentifier [6] to fingerprint the Windows binary code. An example that illustrates this approach is presented in Fig. 5. Basically, it iterates each relocation entry defined in the `.rel.text` in Windows PE files, acquires its in-disk value V_d at offset o_i, then computes a signature value S using the difference between V_d and the base address of the code V_b in disk, namely V_d - V_b. This signature value shall remain a constant for this particular relocation entry. Then it searches the memory to locate the code by probing the value V_m and checking whether its distance to the randomized kernel base address V_x (we aim to find) is S. For instance, as shown in Fig. 5, the S value of $V_m - V_x$ for the first relocation entry is 0x7015d8, and the forth relocation entry is 0x7ca780. Only when all the relocation entries match the S values, does it mean successfully identifying the fingerprints of the code.

With respect to ELF binary, we can notice that not all the relocation entry can be used by CodeIdentifier. Specifically, for the 2nd and 3rd relocation entry, it has the type of `R_386_PC32`, which means the loader/linker shall places the PC-relative 32-bit address of the symbol into the specified memory location. However, when loading them into memory, they have already been updated with the relative addresses and there is no need to patch them. Also, interesting, the code snippet shown in Fig. 5 contains a function tracing disabling patching where instruction "`call c0683c80`" gets patched to NOP "`xchg %ax, %ax`".

Therefore, if we can remove the 2nd and 3rd relocation entry and use the first and forth ones (with type `R_386_32`, which means linker/loader will place an absolute 32-bit address of the symbol into the specified memory location) to compute the signature, we should be able to locate the corresponding kernel code. As such, we use a different approach to compute the signature compared to CodeIdentifier. In particular, instead of computing different signature value for each relocation entry, we compute it using the difference of the value in the memory snapshot V_m and the value in the disk V_d (i.e., $V_m - V_d$). In this way, we will always get a constant value for all the relocation entries, which is the in fact the randomized offset.

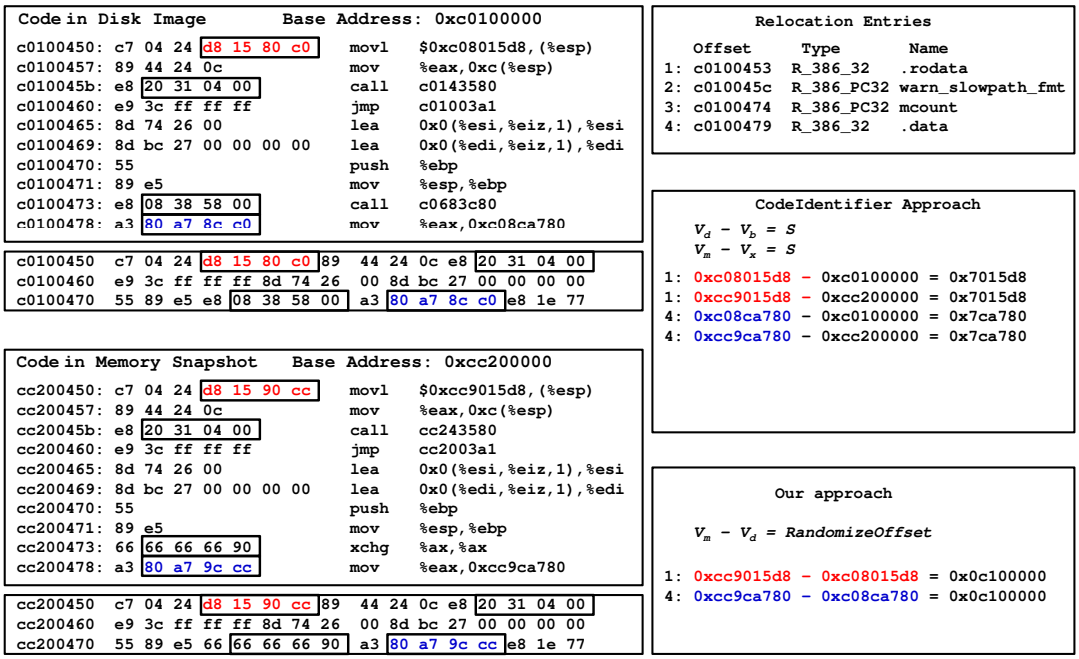

Figure 5: Using Relocation Entry to Generate Code Signatures

For instance, as illustrated in Fig. 5, for the first and forth relocation entries, we will get the same value of 0xc100000.

We also present a rigorous algorithm in Algorithm 2 to illustrate the detailed matching process. Specifically, we probe each possible base address (the maximum is controlled by the n bits entropy used in the randomization) starting from the kernel base address (line 3), if the distance of all the relocation entry point between the randomized kernel and the static disk image (namely $V_m - V_d$) remains a constant value (line 5 - line 10), then we identify and return the randomized base address V_x which is the probed base address (line 12); otherwise, we keep iterating and probing other possible base addresses.

4.3 Unpatched Code Based Approach

Having generated the signatures from patched code for the de-randomization, next we would like to investigate the approaches to generate the signatures from unpatched code. As discussed in §3.2, there are many cases that certain pieces of kernel code can be patched. It is quite challenging to identify the instructions that will not be patched. Meanwhile, we can also notice that we actually do not have to identify all the unpatched code, as long as we can identify the ones that will not be patched and use them as the signatures. That is, we can aim for soundness instead of completeness for unpatched code identification.

Though kernel has complicated cases for load time or run time patching, we realize that all the patching currently only operates individually and there is no dependence between two patched points. For instance, kernel only patches one point at a time based on the information stored in the binary code (e.g., the relocation entry). Therefore, an insight we have is that the instructions that (implicitly or explicitly) increase or decrease stack pointers will not be patched; otherwise, the kernel must patch them simultaneously. For instance, if there is a `push` instructions, there must be a `pop` or equivalent instructions. Otherwise, the stack cannot be kept balanced. This property of stack related instructions keeps themselves from being patched. As a result, they can be served as the candidates for our unpatched code signatures.

Algorithm 3: Unpatched Code Signature Generation

Input: Kernel code C in disk;
Result: A set S which contains all $tuples(\{\text{offset, instruction}\})$ of each unpatched code

```
1 begin
2     S ← ∅ ;
3     for each instruction i ∈ C do
4         if ChangeStackPointer (i) then
5             O_i ← GetOffset (i);
6             C_i ← GetInstrCode (i);
7             S ← S ∪ {<O_i, C_i>}
8     return S;
```

An algorithm of how we generate the unpatched code signature based on the stack operations is presented in Algorithm 3. Basically, we first disassemble the kernel code. Then we collect the instructions that will modify the stack pointers (line 4 - line 7). We consider two categories of instructions in `ChangeStackPointer` function (line 4): one is those implicit stack pointer changing instructions including `push`, `pop` and `leave` that will change the stack size without explicitly modifying the `esp` value; the other ways is the explicit instructions that directly changes the value of `esp`, e.g. "`sub %esp, 0x68`". Our signatures consist of these identified instructions as well as their offsets (line 7, 8).

Then to match the signatures with the memory snapshot, we just probe whether all the instruction at offset $O_i + R$ in memory contains the same instruction C_i as the one in disk O_i. If so, we output R as the randomized offset. The detailed matching algorithms is elided since this is quite simple.

Approach	Total	Signature Generation	Signature Matching	C++	Python
Brute Force	669	0	32	649	20
Patched Code	807	0	110	759	48
Unpatched Code	817	41	107	756	61
Readonly Pointer	822	0	124	773	49

Table 2: Implementation Complexity (Units: LOC).

5. DERIVING SIGNATURES FROM READ-ONLY KERNEL DATA

Data in general can be classified into writable data and read-only data. As discussed earlier, for writable data, when there are pointers involved, the shape of the points-to graph might be able to serve as a signature [18] to derandomize the kernel. We consider this approach is too complicated and instead we focus on read-only kernel data. Meanwhile, for read-only data, strings should be one intuitive candidate for the signatures. However, strings can often be mutated without crashing the kernel. As discussed earlier, while similar to robust signatures approach proposed by Dolan-Gravitt et al. [11], we could use fuzzing to identify those non crashable strings as the signature. Again, we believe this approach is also too sophisticated and leave it for future work. In the following, we discuss a new approach we developed based on the read-only pointers in the `.rodata` section.

5.1 Read-only Pointer Based Approach

Program including OS kernel contains static code pointers, for instance, the system call tables, the indirect jump tables, etc. These static code pointers when compiled are actually stored in the `.rodata` section. Changing the value of these static code pointers will change the semantics of the program code, and it might also lead to the kernel crashes.

Similar to the relocation entries in `.rel.text` that contains those to be patched memory address inside kernel code, there are also relocation entries in `.rel.rodata` that contains the offset in the `.rodata` to inform loader to patch the memory addresses (basically they are pointers) stored in this `.rodata` section. Therefore, we can use them to build read-only data signatures.

The approach is also surprisingly simple. Similar to Algorithm 2, we iterative each relocation entry in `.rel.rodata` (instead of `.rel.text`), and then compare the values between disk version and the memory version. If all entry has the same shifted offset, we return this randomized offset. Detailed algorithm is also elided for simplicity.

6. EVALUATION

We have implemented the four approaches presented in §4 and §5, and these implementation code can be found at github.com/utds3lab/derandomization/. Basically, we implemented each approach with a mixture of C++ and Python. The python code is called by C++ and is used to parse the output from `objdump` and `readelf`. The implementation complexity in terms of lines of code (LOC) for each of approach is presented in Table 2. In particular, we use python to parse the address and offset from the `rel.text` and `rel.rodata` sections produced by `readelf` of the kernel binary code. Meanwhile, we use Python to parse the disassembled code produced by `objdump` and get the offset and instruction of the stack pointer change related instructions as the signatures.

OS Kernels	Brute Force	Patched Code	Unpatched Data	Readonly Pointer
Linux-3.14.8	95.45%	100.00%	100.00%	100.00%
Linux-3.14.11	95.45%	100.00%	100.00%	100.00%
Linux-3.14.30	95.46%	100.00%	100.00%	100.00%
Linux-3.15	95.39%	100.00%	100.00%	100.00%
Linux-3.15.2	95.39%	100.00%	100.00%	100.00%
Linux-3.15.4	95.39%	100.00%	100.00%	100.00%
Linux-3.16	95.40%	100.00%	100.00%	100.00%
Linux-3.16.2	95.40%	100.00%	100.00%	100.00%
Linux-3.16.6	95.40%	100.00%	100.00%	100.00%
Linux-3.17	95.39%	100.00%	100.00%	100.00%
Linux-3.17.2	95.39%	100.00%	100.00%	100.00%
Linux-3.17.6	95.39%	100.00%	100.00%	100.00%
Linux-3.18	95.40%	100.00%	100.00%	100.00%
Linux-3.18.2	95.40%	100.00%	100.00%	100.00%
Linux-3.18.4	95.40%	100.00%	100.00%	100.00%
Linux-3.18.6	95.40%	100.00%	100.00%	100.00%
Linux-3.19	95.40%	100.00%	100.00%	100.00%
Linux-3.19.2	95.41%	100.00%	100.00%	100.00%
Linux-3.19.4	95.41%	100.00%	100.00%	100.00%
Linux-4.0	95.41%	100.00%	100.00%	100.00%
mean	95.41%	100.00%	100.00%	100.00%

Table 4: Match Ratio.

In this section, we present our experimental result. We took 20 Linux kernels from version 3.14 to 4.0 for the evaluation. We first tested the effectiveness of each approach with respect to the testing Linux kernels in §6.1, and then we report the performance overhead of each approach in §6.2.

To obtain the physical memory dumps, we run each of the tested Linux kernels in a VMware Workstation configured with 512M bytes RAM (131,072 pages with 4K bytes each) for the guest OS. After the guest OS has booted up, we took a memory snapshot and used it for the testing. Our host machine has an Intel Xeon CPU with 48G memory, installing a Red Hat Enterprise Linux Workstation 6.5 with Linux kernel 2.6.32.

6.1 Effectiveness

Robustness. To evaluate the robustness of each approach, we use the size of the signatures as the metric, though there could be other metrics to measure the robustness of the signature. Again, the intuition is the longer the signature, the more robust the approach is, because an adversary needs to spend more efforts to cheat the system with longer signatures. The "Total Bytes" column in Table 3 shows the signature size of each approach.

We can notice that brute-force approach has the strongest robustness. On average it contains close to 5.88 Mega-bytes data. It will be extremely difficult to forge such signatures except that an adversary could load multiple copies of the code into kernel memory and cheat this approach. The next strongest signature is the read only pointer based approach. On average it contains 342 Kilo-bytes data. If an adversary wants to cheat this approach, it has to simultaneously patch more than 80 Kilo-bytes pointers. For patched code approach, its signature size is 283 Kilo-bytes. Regarding the unpatched code approach, its signature has only 230 Kilo-bytes on average, and eventually it only keeps on average 3.91% of kernel code in the signatures.

We also reported how many signature bytes per page for each approach in Bytes/Page column in Table 3. Note that these data can provide the statistics with respect to the performance of each approach. Basically, more bytes comparison in a page, the slower the approach will be as we show in Fig. 7. We can notice unpatched code has the least data to compare in a page, whereas brute force approach has an entire page to compare.

OS-kernels	Brute Force		Patched code		Unpatched code		Readonly pointer	
	Total Bytes	Bytes/Page	Total Bytes	Bytes/Page	Total Bytes	Bytes/Page	Total Bytes	Bytes/Page
Linux-3.14.8	5,787,280	4,096	278,156	196	225,632	159	331,956	656
Linux-3.14.11	5,788,560	4,096	278,192	196	225,647	159	332,084	656
Linux-3.14.30	5,802,328	4,096	278,900	196	225,933	159	332,416	656
Linux-3.15	5,793,980	4,096	280,476	198	227,514	160	336,204	659
Linux-3.15.2	5,794,108	4,096	280,480	198	227,514	160	336,208	659
Linux-3.15.4	5,794,940	4,096	280,504	198	227,518	160	336,212	659
Linux-3.16	5,844,284	4,096	281,812	197	229,065	160	340,964	658
Linux-3.16.2	5,846,844	4,096	281,840	197	229,084	160	340,956	658
Linux-3.16.6	5,850,044	4,096	281,916	197	229,213	160	341,068	658
Linux-3.17	5,889,452	4,096	284,832	198	230,785	160	344,240	660
Linux-3.17.2	5,889,324	4,096	284,880	198	230,794	160	344,252	660
Linux-3.17.6	5,894,696	4,096	285,416	198	230,886	160	344,396	661
Linux-3.18	5,929,000	4,096	286,508	198	232,155	160	346,384	662
Linux-3.18.2	5,929,704	4,096	286,516	198	232,159	160	346,448	662
Linux-3.18.4	5,930,280	4,096	286,608	198	232,167	160	346,448	662
Linux-3.18.6	5,931,816	4,096	286,612	197	232,242	160	346,480	662
Linux-3.19	5,977,424	4,096	288,156	197	233,339	159	348,064	662
Linux-3.19.2	5,980,280	4,096	288,216	197	233,466	159	348,104	663
Linux-3.19.4	5,982,136	4,096	288,268	197	233,503	159	348,172	663
Linux-4.0	6,015,102	4,096	289,532	197	235,018	160	351,676	656
mean	5,882,580	4,096	283,891	198	230,182	160	342,137	660

Table 3: Signature Size.

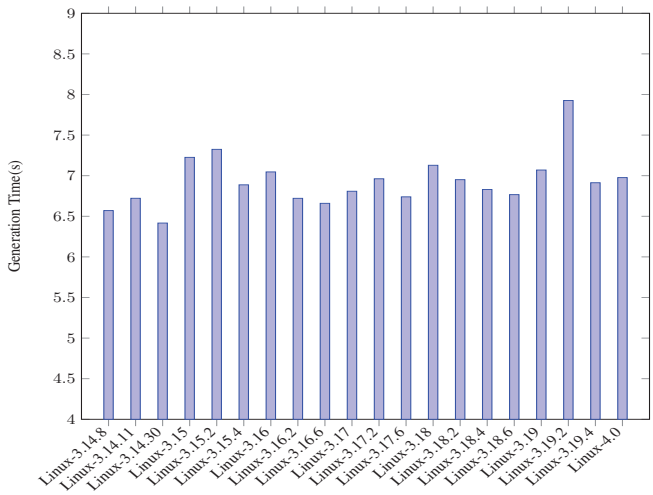

Figure 6: Signature Generation Performance.

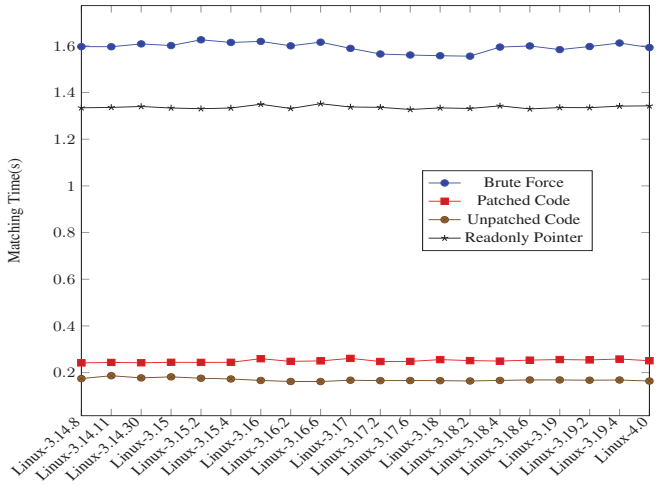

Figure 7: Signature Matching Performance.

Precision. Next, we tested the precision of each approach with benign kernels. Regarding the non-cooperative kernels, it is discussed in §7. The match ratio for each approach is presented in Table 4. We can see that the peak match ratio for brute-force approach is 95.41% (which means usually 4.59% of code has been patched) on average for all these kernels. For all other approaches, it has a perfect 100% match ratio.

6.2 Performance

We also measured the performance of each approach. The faster the performance, the more likely to be used in practice. As mentioned above, we tested 20 latest Linux kernels. The size of the memory snapshot is set to 512M bytes.To load the memory snapshot as well as the kernel code, it took on average 0.677 seconds. For a fair comparison among all these approaches, we exclude the snapshot loading time. Also, only the unpatched code approach requires the preprocessing of the kernel code to generate the signatures, and this performance is presented in Fig. 6 for all these kernels. We can see that on average it took 6.9 seconds for the signature generation. Note that this pre-processing runs with pure python environment, which explains why it took such a larger amount of time than even brute-force matching.

Regarding using the signature for the matching and derandomization, this performance is presented in Fig. 7. We can see that brute-force approach has the worst performance. It took on average 1.6 seconds to derandomize a Linux kernel. The next worst one is the read only pointer based approach, since it needs to check over 85 thousands of pointers and perform the distance subtraction computation for each pointer. The unpatched code is the fastest, and it only requires about 0.2 seconds on average to derandomize a kernel.

7. DISCUSSIONS AND FUTURE WORK

In this study, we aim to systematically examine the possible optimal approaches from both kernel code and kernel data to derandomize kernel ASLR. We have presented four approaches from the perspective of basic brute-force code, to patched code, unpatched code, and read only pointers. However, we have not compared with

a few other possible approaches such as from writable-data, or other read only data to generate the signatures. We leave these to future works.

In our evaluation, we tested our approaches with only benign Linux kernels, and we did not evaluate the memory snapshot running by adversaries such as uncooperative cloud users. The fundamental reason is due to the fact that there is a large attack surface, and it is hard to enumerate all of them. Fortunately, what an adversary they can only play with in the attack is to generate bogus data to mislead the signature matching, because the signatures we used are either code or pointer related, changing them will change the semantics or even crashing the kernel. Therefore, we can analyze the consequences of these non-cooperative attacks.

Specifically, for brute-force code scanning approach, it could fail to derandomize a kernel when directly using the Algorithm 1. For instance, when an adversary simply loads a copy of the kernel code into the kernel memory (by writing a simple kernel module to do so), this approach can get a 100% peak match ratio for this code copy instead of the 95% we tested with the benign kernel. Therefore, even though it has the strongest robustness in terms of signature size, it will have false positive (not providing the correct address) and false negative (missing the correct address) when facing bogus data cheating attack. One possible mitigation would be to relax the peak value to make it more conservative. For instance, through empirical evaluation we can acquire a threshold of the peak value for the benign kernel, and then use it to derandomize the kernel. In this way, it will not have false negatives but it will have false positives since the bogus data will be included.

For all other approaches, while an adversary can also create bogus data, we will not have false negatives since the match ratio is 100%. In other words, the only consequence is that we will not be able to quickly pinpoint the derandomization address but the candidates are among the final results. With additional pruning or a combination with other approaches, it is very likely to produce the correct result. In fact, without too many efforts, we have designed four simple approaches (with just about three thousands lines of code in total), and we believe these four approaches can be combined to prune the false positives (i.e., the bogus data generated by attackers). We leave the investigation for more robust signatures under the presence of strong adversary in another avenue of our future work.

8. CONCLUSION

Many modern OS kernels today have started to randomize their kernel address space. Consequently, when performing introspection or forensic analysis of the kernel memory, we must derandomize and use the correct addresses. In this paper, we present the first systematic study of how to derandomize a kernel when given a memory snapshot of a running kernel instance. Unlike the derandomization approaches used in developing memory exploits in which only remote access is available, with introspection and forensics applications, we can use all of the information available in kernel memory to generate signatures and derandomize the kernel address space layout. We have explored a number of typical approaches to generate strong signatures from both kernel code and data based on the insight of how kernel code and data is updated, and compare them from efficiency and robustness perspective. In particular, we have designed four approaches from brute-force code scanning, to robust signature generation from patched code and unpatched code, as well as from read-only kernel data, respectively. We show that brute-force approach is simple, but it is slow and may have false positives and false negatives to bogus data misleading attack. For all other approaches, they run faster and they will not have false negatives, but there might be false positives when facing strong adversaries. However, these false positives could be pruned when combining with multiple strong signatures for instance.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful comments. This research was partially supported by AFOSR Award FA9550-14-1-0119, NSA Award H98230-15-1-0271, and NSF Award 1453011. Any opinions, findings, conclusions, or recommendations expressed do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] Elf file format. *http://www.skyfree.org/linux/references/ELF_Format.pdf*.

[2] Microsoft security intelligence report. *http://www.microsoft.com/security/sir/strategy/default.aspx#!section_3_3*.

[3] Smp alternatives. *http://lwn.net/Articles/164121/*.

[4] Os x mountain lion core technologies overview. *http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf*, June 2012.

[5] Linux 3.14. *http://kernelnewbies.org/Linux_3.14*, Mar 2014.

[6] I. Ahmed, V. Roussev, and A. A. Gombe. Robust fingerprinting for relocatable code. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015*, pages 219–229, 2015.

[7] H. Y. Aravind Prakash, Eknath Venkataramani and Z. Lin. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN-PDS 2013)*, Budapest, Hungary, June 2013.

[8] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242. IEEE Computer Society, 2014.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proc. 15th ACM Conf. Computer and communications security (CCS'08)*, pages 27–38, Alexandria, Virginia, USA, 2008. ACM.

[10] S. Designer. "return-to-libc" attack. *Bugtraq*, August 1997.

[11] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[12] Y. Fu, Z. Lin, and D. Brumley. Automatically deriving pointer reference expressions from executions for memory dump analysis. In *Proceedings of the 2015 ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'15)*, Bergamo, Italy, September 2015.

[13] Y. Fu, Z. Lin, and K. Hamlen. Subverting systems authentication with context-aware, reactive virtual machine introspection. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*, New Orleans, Louisiana, December 2013.

[14] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In

Proc. Network and Distributed Systems Security Sym. (NDSS'03), February 2003.

[15] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Os-sommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12)*, San Jose, CA, October 2012.

[16] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Multi-aspect, robust, and memory exclusive guest os fingerprinting. *IEEE Transactions on Cloud Computing*, 2014.

[17] T. Kittel, S. Vogl, T. K. Lengyel, J. Pfoh, and C. Eckert. Code validation for modern os kernels. In *Malware Memory Forensics Workshop (MMF)*, December 2014.

[18] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. 18th Annual Network and Distributed System Security Sym. (NDSS'11)*, San Diego, CA, February 2011.

[19] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, 10(58), 2001.

[20] N. A. Quynh. Operating system fingerprinting for virtual machines, 2010. In DEFCON 18.

[21] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, pages 60–69. IEEE Computer Society, Dec. 2009. Honolulu, Hawaii, USA.

[22] V. Roussev. Data fingerprinting with similarity digests. In *Advances in digital forensics vi*, pages 207–226. Springer, 2010.

[23] V. Roussev, I. Ahmed, and T. Sires. Image-based kernel fingerprinting. *Digit. Investig.*, 11:S13–S21, Aug. 2014.

[24] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. 14th ACM Conf. Computer and communications security (CCS'07)*, pages 552–561, Alexandria, Virginia, USA, 2007. ACM.

[25] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[26] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[27] P. Team. Pax address space layout randomization (aslr). *http://pax.grsecurity.net/docs/aslr.txt*.

[28] D. Urbina, Y. Gu, J. Caballero, and Z. Lin. SigPath: A Memory Graph Based Approach for Program Data Introspection and Modification. In *Proceedings of the 19th European Symposium on Research in Computer Security*, Wroclaw, Poland, September 2014.

[29] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.

[30] J. Zeng and Z. Lin. Towards automatic inference of kernel object semantics from binary code. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, Kyoto, Japan, November 2015.