Title

On the Trustworthiness of Memory Analysis—An Empirical Study from the Perspective of Binary Execution

Permalink
https://escholarship.org/uc/item/9700j4xr

Journal
IEEE Transactions on Dependable and Secure Computing, 12(5)

ISSN
1545-5971

Authors
Prakash, Aravind
Venkataramani, Eknath
Yin, Heng
et al.

Publication Date
2015

DOI
10.1109/tdsc.2014.2366464

Peer reviewed

On the Trustworthiness of Memory Analysis—An Empirical Study from the Perspective of Binary Execution

Aravind Prakash, Eknath Venkataramani, Heng Yin, *Member, IEEE*, and Zhiqiang Lin, *Member, IEEE*

Abstract—Memory analysis serves as a foundation for many security applications such as memory forensics, virtual machine introspection and malware investigation. However, malware, or more specifically a kernel rootkit, can often tamper with kernel memory data, putting the trustworthiness of memory analysis under question. With the rapid deployment of cloud computing and increase of cyber attacks, there is a pressing need to systematically study and understand the problem of memory analysis. In particular, without ground truth, the quality of the memory analysis tools widely used for analyzing closed-source operating systems (like Windows) has not been thoroughly studied. Moreover, while it is widely accepted that value manipulation attacks pose a threat to memory analysis, its severity has not been explored and well understood. To answer these questions, we have devised a number of novel analysis techniques including (1) binary level ground-truth collection, and (2) value equivalence set directed field mutation. Our experimental results demonstrate not only that the existing tools are inaccurate even under a non-malicious context, but also that value manipulation attacks are practical and severe. Finally, we show that exploiting information redundancy can be a viable direction to mitigate value manipulation attacks, but checking information equivalence alone is not an ultimate solution.

Index Terms—Memory forensics, operating systems security, invasive software, DKOM, kernel rootkit, virtual machine introspection

1 Introduction

Memory analysis aims at extracting the semantic knowledge from a memory snapshot or a live memory of a running computer system. Such extraction has been proven to be valuable for various computer security problems such as memory forensics (for memory snapshot) and virtual machine introspection (VMI) (for live memory). For instance, by examining a memory image, we could extract semantic information about all of the running processes, open files, and network connections.

Digital forensics collects evidence of digital crimes, intrusions, and malware attacks from a victim's computer system. Although criminals tend to avoid leaving any evidence in the persistent storage, it is extremely hard for them, if not impossible, to completely remove their footprints in memory. Therefore, memory forensics becomes increasingly irreplaceable. With the knowledge of kernel data structures, tools such as FATKit [1] and Volatility [2] can parse a memory image (either a snapshot or a hibernation file), traverse the kernel data structures, and extract semantic information for human analysis.

Virtual machine introspection is a technique used by the virtual machine monitor to reconstruct an out-of-the-box semantic view of a virtual machine, in order to analyze, detect, and prevent intrusions and malware attacks [3], [4], [5], [6]. Also, VMI is one of the enabling techniques for cloud security and management. To bridge the semantic gap [7]-a hard problem in VMI, the virtual machine monitor often has to extract the semantic knowledge from live memory or a memory snapshot of the virtual machine. From a memory analysis point of view, VMI and memory forensics share the same flavor in that they both have to reconstruct the semantic state of a memory image.

On the flip side, to thwart memory analysis, kernel rootkits, once having penetrated into an operating system kernel, can arbitrarily alter the kernel code and data. Particularly, a family of kernel rootkits (e.g., FU Rootkit [8]) can directly tamper with the kernel data structures, also known as Direct Kernel Object Manipulation (DKOM) attacks. Consequently, the trustworthiness of memory analysis becomes questionable, which directly hurts memory forensics and VMI.

Some recent efforts have been made to improve the robustness of memory analysis. For instance, Dolan-Gavitt et al. proposed a more robust field-based signature scheme [9]. Specifically, for each data structure of interest, this scheme identifies a minimum set of value patterns that cannot be tampered by attackers. Otherwise it may render system malfunction or crash. As a result, hidden objects can be identified with more confidence. Meanwhile, since not all kernel data structure fields could have a field-value based robust signature, Lin et al. proposed a complementary scheme, a graph-based signature built on field points-to relation to reduce false positives (FP) and false negatives (FN), at the price of higher scanning performance overhead [10] because of the pointer traversal. Although these signature approaches can identify objects of interest with higher

- A. Prakash, E. Venkataramani, and H. Yin are with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244. E-mail: {arprakas, evenkata, heyin}@syr.edu.
- Z. Lin is with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75080. E-mail: zhiqiang.lin@utdallas.edu.

Manuscript received 18 Oct. 2013; revised 14 Oct. 2014; accepted 20 Oct. 2014. Date of publication 30 Oct. 2014; date of current version 16 Sept. 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2014.2366464

confidence, the trustworthiness of the semantic values in these objects are still under question.

While memory analysis serves as a foundation for many important security applications, the *dependability* of memory analysis, remains unclear to the security community. We aim to conduct first such study. In particular, we focus on kernel memory analysis, as the operating system semantics are crucial for security. User-level software memory analysis, such as game cheating and map hacking [11] or browser-related activities, is out of scope of this paper.

More concretely, we conduct a systematic empirical study on the dependability of kernel memory analysis under both honest and deceptive contexts. In the honest context where the digital evidence in the kernel is not manipulated, we aim to evaluate the correctness of the existing memory analysis tools. This is important especially for the closed-source operating systems like Windows, because these memory analysis tools are built based on limited knowledge about the closed-source systems and thus may be error prone.

A key challenge for conducting such an empirical study is how to obtain the ground truth. While for an open source OS, one could instrument the source code to collect the ground truth, the real difficulty is for a closed source OS for which we have to perform reverse engineering. To this end, we take a unique perspective from *dynamic binary analysis*. That is, we monitor and analyze the binary execution of the OS. Since the OS binary directly executes on the hardware, we believe that the semantic knowledge extracted from the execution provides the ground truth (at least faithful for this particular execution). Moreover, with dynamic binary analysis, we are able to evaluate memory analysis tools not only on open source OS, but also on closed-source OS such as Microsoft Windows. For this study, we particularly evaluated the efficiency and accuracy of the analysis tools in the Volatility memory analysis framework [2] and two robust signature schemes [9], [10].

In the deceptive context where the digital evidence may have been manipulated and/or destroyed, we aim to evaluate the trustworthiness of the existing memory analysis tools. As prior efforts have been made to defeat pointer manipulations [10], in this paper, we focus on evaluating *Semantic Value Manipulation* (SVM) attacks, which directly manipulate data values in kernel data structures to mislead security tools. Therefore, we assess the trustworthiness of kernel memory analysis by discovering the attack space and the severity of SVM attacks. On one hand, with the highest privilege, an attacker can modify arbitrary memory locations; on the other hand, she does not want these modifications to introduce noticeable differences in system behavior (e.g., crash, instability, and malfunction).

Systematic and calculated SVM is a challenging task. A brute force approach would be to fuzz all the kernel variables one at a time, but tends to be inefficient. To demonstrate the severity of the SVM attacks and explore the possible defense approaches, we devise a new fuzzing technique to mutate the exact fields pinpointed by our analysis. We have implemented a prototype system, named *MOSS* (short for "Mutating OS Semantics") that has two unique features: (1) It is *semantic-field oriented*, meaning it can cooperate with the test program and automatically locate the data structure fields that hold specified OS semantic information and mutate their values; and (2) it is *duplicate-value directed* because semantic values are often duplicated in various data structures.

To further demonstrate the attack impact, we implemented a proof-of-concept kernel rootkit, based on FUTo [8] (a DKOM rootkit for Windows). Specifically, we installed a real-world bot, TDSS [12] in a controlled Windows XP guest OS and using the rootkit, we performed simultaneous mutations to all vulnerable semantic fields identified by MOSS. The mutations were targeted at hiding and/or misleading the state-of-the-art security tools without leading to a program or a system crash.

Through the empirical study, we make the following observations:

- The accuracy of basic memory analysis tools (including traversal-based and signature-based ones) is not perfect even under a non-malicious context.
- SVM attacks are practical, and many semantic values that are critical to security investigation can be altered without causing adverse effect to the victim system.
- Exploiting information redundancy could be a viable solution to detect value manipulation attacks. However, simply checking duplicate values is not a complete solution to this problem.

This paper is an extended version of the work published in [13]. Specific extensions include examination of memory analysis tools under a deceptive and non-deceptive context (in Sections 2 and 3) and a root-cause analysis on why SVM attacks succeed (in Section 4.5).

2 Questions for Our Empirical Study

Examining the memory analysis tools. We aim to examine the quality and performance of memory analysis tools that retrieve information pertaining to processes, modules, IO, files, etc. Specifically, we attempt to evaluate the Volatility memory analysis framework with many of its analysis plugins. These plugins can be classified into traversal-based and signature-based schemes. For example, `pslist` is a traversal-based plugin that traverses the active process list and prints the running processes, whereas `psscan` is a signature-based plugin that scans the entire memory for `EPROCESS`—process objects. Therefore, we can evaluate both traversal-based and signature-based schemes using the Volatility framework. Thus, we have the following two questions:

Q1. How accurate and efficient are the traversal-based tools under non-deceptive context?

Q2. How accurate and efficient are the signature-based tools under non-deceptive context?

Examining robust signature schemes. Robust signature schemes, including field-based SigField [9] and graph-based SigGraph [10], can be used to defeat value and link manipulation attacks. We aim to evaluate the quality and performance of these robust signature schemes.

Q3. How accurate and efficient is the robust field signature scheme under non-deceptive context?

Q4. How accurate and efficient is the graph signature scheme under non-deceptive context?

Exploring value manipulation attacks. The power of value manipulation attacks have not been extensively studied in the literature. This time, we play the role of an attacker to explore the attack space.

Q5. How severe can value manipulation attacks be?

Exploiting the information redundancy (especially value equivalence) in the kernel can be a potential solution to this attack. To efficiently explore this potential, we devise a dynamic binary analysis technique to identify these value equivalence sets during the kernel execution. Having identified the equivalence sets, we employ a *value equivalence set directed mutation* technique to *efficiently* examine whether attackers can completely manipulate the values. In particular, we want to investigate the following question:

Q6. To what degree can value equivalence checking help detect value manipulation attacks?

3 Evaluating Memory Analysis Tools in Non-Deceptive Context

In this section, we describe how we evaluate the most common memory analysis tools with respect to Q1-Q4, in a non-deceptive context. We first present how we collect the ground truth—the most crucial part for our analysis—in Section 3.1, then describe how we set up our experiment in Section 3.2, followed by the evaluation of the basic traversal and signature schemes, and robust signatures in Sections 3.3 and 3.4, respectively.

3.1 Constructing the Kernel Data Structure Graph (DSG)

Before we begin our evaluation, we first acquire the ground truth, namely the true kernel objects. This is a challenging task especially for closed source OS. To make our analysis more general, we take a *dynamic binary analysis* based approach to collect the ground truth.

Kernel objects exist in the kernel data structure graph, and they can be reached from either kernel global variables, stack variables, or registers. Meanwhile, since the kernel can not statically anticipate the number of required kernel objects, and kernel stack for each process usually has limited size (typically between 4 and 8K), it tends to dynamically allocate kernel objects. Thus, to our knowledge and also from our experiments, all kernel objects of interest are dynamically allocated.

Therefore, to acquire our ground truth, we implemented a plugin for DECAF [14], to monitor the execution of an operating system and construct the kernel data structure graph on the fly. To construct this graph, we need to monitor the lifetime of kernel objects and recognize all the links between these kernel objects. Unlike static analysis which has to resolve the points-to relationship between pointers, we do not have to perform any such analysis because in dynamic analysis we have all the values and we directly link the points-to data based on the values.

To monitor the lifetime of kernel objects, we hook the allocation and deallocation routines in the kernel. For Windows, `ExAllocatePoolWithTag` is used to allocate an object from a dynamic pool, and `ExFreePoolWithTag` deallocates a kernel object. We hook these two functions to keep track of live objects in the kernel. In addition, we also need to keep track of kernel modules, because global data variables and pointers are located in the memory regions of these kernel modules. The memory map of these kernel modules is derived by hooking `MmLoadSystemImage`. These functions are located in the main kernel component `ntoskrnl.exe`, and the offsets of the functions can be obtained from the symbol information. Since `ntoskrnl.exe` is always loaded at the same base,[1] the entry points for these functions can be determined in advance.

To accurately recognize the links between kernel objects, we make use of dynamic taint analysis, a variant of widely studied data flow analysis in security. When a new object is allocated, we mark the return value of the allocation routine as *tainted*. In other words, we taint the *root* pointer of the newly created object. By keeping track of taint propagation, we can correctly identify the pointer fields that actually point to this object. In many cases, a pointer field in an object may point to the middle of another object. To determine this offset, we store the base address of a newly created object into its taint tag, and when this tag propagates to a pointer field, we can subtract the actual value in the pointer field by the base address in the tainted tag to get the offset.

Moreover, a pointer field may also refer to a label in a static data region of a kernel module. To recognize these labels and taint them, we perform static analysis on the kernel modules. We leverage the fact that the kernel modules in Windows (as well as Linux) are actually compiled as relocatable, and these labels have to appear in the relocation tables in these kernel modules. Thus, we can iterate through the relocation tables and then mark these labels as tainted.

In addition to monitoring of the kernel objects, we also attempt to infer their types. While an object is being allocated, we examine the call stack of the memory allocation routine to determine the context under which the object is allocated. More specifically, we check the return addresses (experimentally, we found three callers are sensitive enough to distinguish different object types) and the object size on the call stack. The underlying rationale is that programs tend to create the same type of objects under the same context with the same size. By checking three callers on the stack and object size, we are confident that the objects allocated with the same callers and size are of the same type.

3.2 Experiment Setup

Our evaluation in this section mainly aims to study the plugins of Volatility framework, a widely used memory forensics tool. To evaluate their correctness, we run an operating system on top of DECAF and analyze its execution from startup and generate the kernel data structure graph on the fly. After the operating system boots up, we launch several common tests, such as editing in `notepad`, visit webpages in the browser, etc. Then, we pause the virtual machine and dump the physical memory of the virtual machine along with the corresponding kernel data structure graph. We analyze the memory dump using the tools in Volatility and compare the results with our kernel data structure graph obtained from binary analysis.

[1] In Windows 7, due to address space layout randomization, the base address of `ntoskrnl.exe` is randomized. However, as it is loaded in the earliest boot stage, its base address can be obtained at runtime.

TABLE 1
Efficiency of Volatility Tools

Commands	WinXP-SP3		Win7-SP0	
	512 M	1 G	512 M	1 G
`pslist`	1.18 s	1.16 s	1.84 s	1.97 s
`pstree`	1.79 s	1.76 s	3.15 s	3.27 s
`handles`	8.50 s	8.51 s	14.04 s	12.01 s
`dlllist`	1.58 s	1.60 s	2.98 s	3.55 s
`sockets`	1.18 s	1.20 s	-	-
`connections`	1.22 s	1.35 s	-	-
`getsids`	1.26 s	1.27 s	2.07 s	2.89 s
`inspectcache`	0.47 s	0.47 s	0.47 s	0.52 s
`memmap`	32.27 s	29.41 s	1:08.21 s	24.81 s
`modules`	1.22 s	1.23 s	1.91 s	2.15 s
`printkey`	3.00 s	3.01 s	3.50 s	4.62 s
`ssdt`	3.06 s	3.05 s	5.64 s	5.43 s
`vadinfo`	6.77 s	6.11 s	16.68 s	14.35 s
`vadtree`	2.64 s	2.35 s	5.22 s	4.76 s
`vadwalk`	3.42 s	3.04 s	6.65 s	6.01 s
`psscan`	1.76 s	2.27 s	2.36 s	14.81 s
`modscan`	2.48 s	2.36 s	2.52 s	15.75 s
`sockscan`	1.68 s	2.27 s	-	-
`connscan`	1.78 s	14.48 s	-	-
`filescan`	4.02 s	4.21 s	6.55 s	17.60 s
`driverscan`	1.79 s	3.49 s	2.55 s	15.18 s
`thrdscan`	2.10 s	2.71 s	3.20 s	15.65 s
`hivelist*`	1.15 s	2.38 s	1.85 s	14.98 s
`hivescan`	0.45 s	0.46 s	0.47 s	0.78 s
`imageinfo`	1.28 s	17.79 s	1.64 s	79.78 s
`mutantscan`	1.87 s	2.48 s	2.75 s	15.40 s
`netscan`	-	-	4.05 s	41.55 s
`kdbgscan`	0.45 s	0.45 s	0.50 s	0.47 s
`kpcrscan`	507.19 s	1079.42 s	0.50 s	1068.45 s

* *`hivelist` combines signature scan and traversal. It first scans for a `hive` object and then traverses the hive linked list from there.*

TABLE 2
Accuracy of Volatility Tools

Commands	WinXP-SP3				Win7-SP0			
	DSG	Vol.	FN	FP	DSG	Vol.	FN	FP
`pslist`	22	22	0	0	32	32	0	0
`psscan`	22	22	0	0	32	32	0	0
`pstree`	22	22	0	0	32	32	0	0
`sockets`	21	21	3	3	-	-	-	-
`sockscan`	21	22	0	1	-	-	-	-
`connections`	6	6	0	0	-	-	-	-
`connscan`	6	6	0	0	-	-	-	-
`filescan`	1,586	1,807	6	0	2,709	2,398	0	0
`driverscan`	74	74	0	0	106	100	0	0
`thrdscan`	325	326	0	1	413	422	0	9
`mutantscan`	149	149	0	0	258	258	0	0
`netscan`	-	-	-	-	68	70	0	2

We conducted this experiment for two Windows OS releases, namely Windows XP Service Pack 3 and Windows 7 Service Pack 0, both of which have more than tens of millions of users.

3.3 Evaluating Basic Schemes: Q1 & Q2

3.3.1 Efficiency of Volatility Tools

Table 1 lists the performance of the Volatility tools. For each of the two operating systems (XP and Windows 7), we created two memory dumps: the first one with 512 MB physical memory, and the second one with 1 GB physical memory. Then, we ran the analysis tools on the four memory dumps and measured their runtime performance. In Table 1, we grouped all traversal-based tools in the upper portion, and signature-based tools in the lower portion. Note that some tools are only available for one OS version.

As expected, the traversal-based tools are in general fairly efficient. Most of the traversal-based tools can provide analysis results in a few seconds. Moreover, their performance is not affected by the size of memory dumps. Surprisingly, many signature-based tools are also very efficient. Intuitively, a sequential scan of 512 MB or 1 GB memory image takes more than a few seconds. After examining the source code, we found that several optimizations are in place to improve the scan efficiency. For example, with the knowledge of heap management in Windows, `psscan` enumerates all the heap objects and looks for heap objects with size of `EPROCESS`, and then applies object specific constraints to further filter them. While these optimizations may work well enough in non-malicious context, they can be exploited by attackers to evade detection. For example, to evade the optimized `psscan`, one may slightly change the object size, and just before the process exits, recover the object size to ensure proper functioning of OS heap management. Furthermore, the performance of several signature-based tools (such as `imageinfo` and `connscan`) degrades significantly while the size of memory dump increases from 512 MB to 1 GB, which is expected for signature-based tools.

3.3.2 Accuracy of Volatility Tools

To evaluate the accuracy, we measured the false positives and false negatives in terms of the number of kernel objects that are checked. As discussed in Section 3.1, objects allocated under the same context with the same size are believed to be of the same type. Therefore, by comparing the objects in the data structure graph and the objects recognized by Volatility, we can establish a mapping from (caller_list, size) to object type. For example, from dynamic analysis, we find an object is allocated at callers $c1 : c2 : c3$ with the size s_1, and this object is then recognized as `EPROCESS` in Volatility, then we believe that all objects allocated at the same caller list and size are `EPROCESS` objects. Note that multiple pairs of (caller_list, size) may be mapped to one object type. Thus this is a many-to-one mapping. Considering that Volatility may wrongly identify an object's type, we have to verify these mappings. We manually look up these callers in the Windows kernel binary using IDA Pro. By examining the disassembly annotated with public symbols, we found it relatively straightforward to confirm whether a mapping is correct. We found one mapping for `_DRIVER_OBJECT` and one for `_FILE_OBJECT` to be erroneous and were excluded.

We examine some common and relatively simple Volatility plugins to evaluate their accuracy. As a first step, we obtain an image along with the ground-truth for the image obtained via dynamic memory analysis. Then, we subject the image to Volatility plugins and compare the results

TABLE 3
Efficiency and Accuracy of Robust Signature Schemes

Tool	512 MB			1 GB		
	Time	Objs	FN/FP	Time	Objs	FN/FP
SigField	641 s	25	0/0	1,283 s	28	0/0
SigField_opt	341 s	25	0/0	715 s	28	0/0
SigGraph	494 s	25	0/0	1,006 s	28	0/0

against ground truth. The results are tabulated in Table 2. Signature-based tools (including sockscan, thrdscan, and netscan) are observed to have some false positives. We also observed some false negatives—321 for filescan and 6 for driverscan—in Windows.

After taking a closer look at the false positives and false negatives, we found that out of the 321 file objects, 315 have blank file names or have OBJECT_HEADER.Type == 0xBAD0B0B0, a sentinel value used by Windows to flag objects that have been deleted. The remaining six file objects were false negatives resulting from a bug in Volatility. The missing six driver objects were also verified to be marked for deletion. We believe that these objects are not actively used at this moment. Therefore, we do not treat them as false negatives.

In reality, the question if or not to capture deleted objects could be requirement dependent. For example, in digital forensics, historic information may be valuable. However, virtual machine introspection and malware analysis generally rely on fresh and up-to-date semantic knowledge thereby eliminating the need for such stale information. Moreover, our ground truth retrieval technique can be easily modified to track the deleted objects in the kernel to provide an analyst with a historic view of the kernel objects however, that deviates from the goal of our work.

Files missed by volatility. We found some active files that were missed by the filescan plugin of Volatility on Windows XP SP3. An object allocated through ExAllocate-PoolWithTag contains a POOL_HEADER object that contains the Tag information associated with the memory pool along with the size of the allocation. POOL_HEADER may be followed by some optional headers specific to the object instance followed by the OBJECT_HEADER and finally the object itself (in case of file object, FILE_OBJECT). Volatility assumes that there is no *gap* between the end of FILE_OBJECT structure and the end of the allocation unit. While this is true for most cases, we found cases when this is not necessarily true. Specifically, when OBJECT_HEADER. Flags has a value 0x66 and the object is allocated in pool type NonPagedPoolCacheAligned, there exists a *pool block* size alignment padding between the end of FILE_OBJECT and the end of allocation unit, due to which Volatility misses the object. This issue, along with a patch to fix the problem has been reported to Volatility.

3.4 Evaluating Robust Signature Schemes: Q3 & Q4

With the kernel data structure graph obtained from dynamic binary analysis, we further evaluate the efficiency and accuracy of two recent robust signature schemes, namely SigFeild [9] and SigGraph [10]. The signature for EPROCESS of SigField in Windows XP is available in Volatility. Thus we directly evaluated this tool. For the original SigGraph scheme, it requires the access to the OS source code and there is no graph-based signature available for Windows. In order to evaluate its quality and efficiency and compare against other tools, we manually created a graph-based signature for EPROCESS in Windows XP SP3 by following their algorithm [10]. This is actually feasible because the definitions for EPROCESS and the structures it points to are publicly available.

Using the same evaluation strategy in Section 3.3, we evaluate the two signatures for EPROCESS and list the results in Table 3. We can see that although the accuracy of these two robust signatures on EPROCESS is perfect, their performance is significantly worse than the basic analysis tools (e.g., pslist and psscan). SigGraph is supposed to be slower, because SigGraph examines pointer and excludes non-pointer fields in objects up to several layers, whereas SigField only checks a few values in each object. Surprisingly, our experiment shows that SigField is actually slower than SigGraph. With further investigation of its source code, we found that SigField makes expensive get_obj_offset() function call frequently to obtain offsets within EPROCESS structure. In comparison, our implementation of SigGraph uses hardcoded offsets. To make a fair comparison, we optimized SigField implementation to use hardcoded offsets. The performance of optimized version (SigField_opt) is better than SigGraph, as expected.

Clearly, these robust signature schemes cannot be directly used for the security applications (e.g., virtual machine introspection) that need to obtain semantic information in a timely manner.

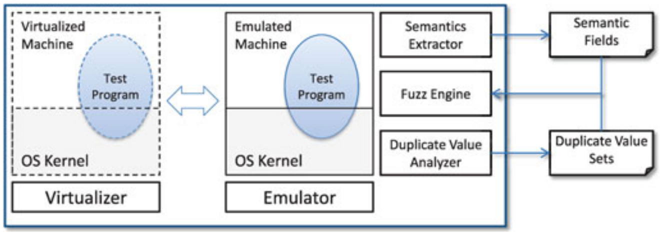


Fig. 1. Architecture of dupcliate-value directed fuzzing.

4 Exploring Semantic Value Manipulation Attacks

In this Section, we aim to evaluate the value manipulation attack, i.e., Q5 & Q6, from an offender's perspective.

4.1 Our Techniques

Fig. 1 illustrates an overview of our fuzz testing system. We run the OS of interest within DECAF [14], a whole-system binary analysis platform. Such a virtualized testing environment facilitates fuzz testing for several reasons. First, it is simple to modify arbitrary memory values. Second, it can easily revert the virtual machine to the previously saved state to conduct fuzz testing in the next round. Last and most importantly, it can dynamically switch between emulation and virtualization mode during testing. In the emulation mode, we perform fine-grained binary analysis to locate duplicate semantic values, and then switch to the virtualization mode to fuzz these duplicate values for better testing efficiency.

More specifically, inside the virtual machine, we run a test program to activate the kernel side execution. Note that we are mutating the semantic values that are related to malicious activities. That is, the attacker attempts to manipulate semantic values pertaining to malicious behavior, such as the name of the malicious process, the file that has been accessed, and so on. These malicious activities are often stealthy and have infrequent interactions with the victim system. To mimic these malicious activities, our test program does not need to achieve the high test coverage of the OS kernel code. Instead, our test program just need conduct some common tests to exercise different OS subsystems, such as task management, file system, network stack, etc. Therefore, if all the mutation attempts on a semantic value do not cause adverse effect in these test cases, we can conclude that such a semantic value is mutable. Otherwise, if a semantic value is sensitive enough to all the mutation attempts on it, we have confidence that this semantic value is immutable and thus tend to be trustworthy. From an attacker's standpoint, some semantic values are in between: some mutations cause system instability while some others do not. These semantic values are *partially* mutable. Note that partially mutable semantics are pertinent in an attack scenario and may be fully mutable under a different context (e.g., as a result of legitimate kernel functionality).

On top of DECAF, we develop three components: semantics extractor, fuzz engine, and duplicate value analyzer. The semantics extractor, which will be discussed in Section 4.1.1, locates the semantic values from the memory snapshot of the guest system. The duplicate value analyzer monitors the kernel execution and performs dynamic duplicate value analysis, which will be detailed in Section 4.1.2. At a high level, it clusters the memory locations into sets, each of which holds the same semantic value. The fuzz engine coordinates with the other two components to conduct automated fuzz testing, which will be discussed in Section 4.1.3.

4.1.1 Locating Semantic Values

At a certain execution point, we need to locate the semantic values to be mutated. Semantic values for mutation are selected in cooperation with the test program inside the virtual machine. A test point has been defined within the test program, dictating which semantic value or which set of values need to be mutated. More details will be discussed in Section 4.1.3. Then, the semantic extractor needs to locate the selected semantic value in the guest kernel memory space.

We leveraged Volatility memory forensics framework [2] and implemented a plug-in to locate the semantic values of interest. More specifically, at the test point, the virtual machine is paused, and a memory snapshot is taken. Then our Volatility plug-in will parse the kernel data structures in the memory snapshot and identify both virtual and physical addresses for the selected value. The virtual address will be used as input to find duplicate value sets, which will then be mutated individually and simultaneously in the subsequent fuzz testing.

4.1.2 Dynamic Duplicate Value Analysis

Many memory locations share the same value at a given moment, either coincidentally, or because of program logic. Our interest is in the latter case since such duplicates hold values which have the same semantic meaning. We call these variables *true* duplicates. To identify true duplicate values, we devise a dynamic binary analysis algorithm that classifies variables (memory locations or registers) into clusters. Variables belonging to the same cluster hold the same semantic value because of the program logic in this particular program execution.

TABLE 4
Algorithm Execution on the Sample Code

Statement	S_a	S_b	S_c	S_d	S_e	S_f
`1: a=b`	{a,b}	{a,b}				
`2: c=a`	{a,b,c}	{a,b,c}	{a,b,c}			
`3: d=b+c`	{a,b,c}	{a,b,c}	{a,b,c}			
`4: e=a`	{a,b,c,e}	{a,b,c,e}	{a,b,c,e}		{a,b,c,e}	
`5: b=2`	{a,c,e}		{a,c,e}		{a,c,e}	
`6: f=c`	{a,c,e,f}		{a,c,e,f}		{a,c,e,f}	{a,c,e,f}

To better explain the idea of dynamic duplicate value analysis, consider the example code in Table 4. After executing the six statements under "Statement" column of Table 4, variables a, c, e, and f should have the same value, so these variables should belong to the same cluster. b belongs to this cluster till line 5, where b is assigned to a different value. Suppose that e is identified to have a semantic meaning such as `pid` of a process, we can conclude that the other variables (a, c, and f) in the same cluster should also hold the `pid` of that process. Therefore, we need to perform dataflow analysis to compute these clusters.

Yet, the existing forward dataflow analysis (i.e., taint analysis [15]) and backward dataflow analysis (i.e., backward slicing [16]) cannot solve this problem. For taint analysis, the taint source needs to be known in advance. However, in our case, semantic values can only be identified at a later stage. Backward slicing is not a solution either. Starting from line 4 and walking backward the code snippet, backward slicing can identify e is directly copied from a and b, but c and f are missing. Moreover, b should not be a redundant value, because b is later assigned to a different value at line 5. To solve this problem, we devise a new dynamic dataflow analysis algorithm called *dynamic duplicate value analysis* to compute the clusters at runtime. The basic algorithm is shown in Algorithm 1.

The idea behind our Algorithm 1 is as follows. At memory byte granularity, we treat each memory byte as a variable r and a redundancy cluster S_r is associated with each variable r. Based on each instruction's semantics from the execution traces, we perform data flow analysis. More specifically,

- *Direct assignment.* For each instruction i in the execution trace, we check if i is an assignment operation. In x86, assignment operations include `mov`, `push`, `pop`, `movs`, `movzx`, `movsx`, etc. As a variable represents a memory byte, we break an assignment into one or more per-byte assignments, and for each source and destination byte pair (u, v), we update the duplicate sets accordingly (as shown in `DoAssign`). First of all, the destination v is no longer equivalent to the other variables r in its old duplicate set S_v, and thus v needs to be removed from S_r. Furthermore, since v is now equivalent to u, v also needs to be added into the duplicate set S_r, where $r \in S_u$. Lastly, the duplicate set of v will be updated to that of u. In general, a membership change of a variable in its duplicate set needs to propagate around to maintain consistent membership information. SSE and MMX instructions may also serve as data transfer operations. We do not consider these instructions because we found in our experiments that these instructions rarely appear in the kernel execution.

- *Other operations.* For the rest of the instructions, while the duplicate sets for the source operands remain the same, the duplicate set for the destination operand needs to be reset. Therefore, for each byte v of the destination operand, DoRemove notifies all variables in v's duplicate set that v is no longer their duplicate.

Table 4 gives a step-by-step demonstration of how the algorithm executes on the sequence of statements.

Algorithm 1. Dynamic Duplicate Value Analysis

```
Procedure DynValueAnalysis (Trace $t$)
  for all instruction $i \in t$ do
    if $i.type$ is assignment operation then
      for each src & dst byte pair$(u, v)$ do
        DoAssign$(u, v)$
      end for
    else
      for each byte $v$ in the dst operand do
        DoRemove$(v)$
      end for
    end if
  end for
end procedure

procedure DoAssign $(u, v)$
  for all variable $r \in S_v$ do
    $S_r \leftarrow S_r - \{v\}$
  end for
  for all variable $r \in S_u$ do
    $S_r \leftarrow S_r + \{v\}$
  end for
  $S_v \leftarrow S_u$
end procedure

procedure DoRemove$(v)$
  for all variable $r \in S_v$ do
    $S_r \leftarrow S_r - \{v\}$
  end for
end procedure
```

Extension for string conversions. Algorithm 1 only handles literal value equivalence. For strings, the operating system kernel often makes conversions, such as from ANSI to UNICODE or vice versa, or from upper case to lower case or vice versa. Semantically, a converted string is equivalent to the original string. Therefore, we have to extend the basic algorithm to maintain the equivalence relation between the converted and original strings. We hook the string handling functions in Windows and directly call DoAssign to make the duplicate value association between the input and the output.

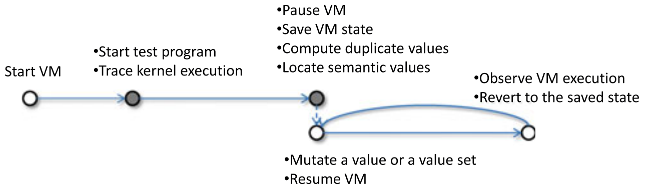


Fig. 2. Fuzz Testing Cycle. A gray node indicates the virtual machine at that moment is running in the emulation mode, whereas a white node stands for the virtualization mode.

Discussion. Through direct assignments and restricted string conversion, our algorithm captures how normal program execution operates on duplicate values. Thus, it is able to correctly identify duplicate values in regular programs. However, a program may be obfuscated to evade our analysis. As an example, a direct assignment can be replaced by a sequence of arithmetic or logic operations. As we apply this algorithm to benign kernel code analysis, this limitation does not apply.

Moreover, as a dynamic analysis technique, the identified duplicates depend on the program execution. In our setting, we trace the kernel execution from the start of the test program to a designated test point, so the creation and propagation of the semantic values associated with the test program should be completely captured and analyzed.

4.1.3 Testing Procedure

Testing cycle. As depicted in Fig. 2, a testing cycle proceeds as follows: (1) In the virtualization mode, start the virtual machine and boot up the guest system. (2) Switch to emulation mode, run the test program and start to trace kernel execution for duplicate value analysis. (3) At a predetermined test point, pause the virtual machine and save the current VM state; in the meantime consult the semantics extractor to locate important semantic values and query the duplicate value analyzer to compute duplicate value sets; and switch to the virtualization mode. (4) Choose to mutate a single value or a set of duplicate values, and resume the virtual machine; (5) The test program finishes normally or prematurely or system crashes; revert to the saved VM state and go to step 4 to fuzz another semantic value or another duplicate value set. We define multiple test points to mutate different sets of semantic values and conduct the above test cycle multiple times, once for each test point.

Test program. We design our test program to exercise basic and common operations that are commonly performed by programs and that are typically exhibited by malware. Various test points and test cases exercised by the test program are enumerated in Table 5. We can see that the test program exercises process and thread management, DLL load and unload, kernel module management, file operations, network operations, and registry key accesses (for Windows only). In all, six test points are defined at precise moments, when the virtual machine will be paused and selected semantic values will be mutated. These test points capture the moment when certain kind of values have been created and will be used for later operations. For example, for file related semantic values, the test point is defined after the files are open and before read and write operations are performed on these files. Furthermore, in order to prevent

TABLE 5
Test Cases and Test Points

No	Test Case
1	Start test program Test Point 1: mutate process & thread related values Run other test cases
2	Load a user DLL Test Point 2: mutate DLL related values Call a function in the DLL repeatedly Unload the DLL
3	Load a kernel module Test Point 3: mutate kernel module values Send IO requests to the kernel module Unload the kernel module
4	Open two files, one each for read and write Test Point 4: mutate file values Read and write the two files repeatedly close the files
5	Open a TCP connection Test Point 5: mutate values related to this connection Send and receive data through this connection Close the connection
6	Open a registry key (Windows only) Test Point 6: Mutate registry key related values Read and write this registry key repeatedly Close the key

reverting of the VM from interfering with the HTTP session for network related tests, we launch a light weight HTTP server on the guest OS where tests are conducted.

The identified test points are tested individually. For instance, when we conduct fuzzing on the first test point, the other test points are simply skipped, however the test program continues to perform all the operations listed in Table 5 during all the tests. This is important since a change in a test point could have an implication in multiple functionalities. For instance, a change to a thread related semantic value might result in dropping of the connection that the thread has made. Also, the order of the test cases listed in Table 5 does not reflect the actual order of our fuzz testing. Suppose that we are conducting test case 5 for the network connection. We actually move this test case earlier, immediately after the test program starts, such that we can observe if the mutation of network-related semantic values will affect the execution of the other test cases.

Mutation rules. To avoid system instability due to mutation, the changes have to satisfy the type constraint of the original value. In other words, the mutation rules depend on the type of the semantic value to be mutated. In contrast, other fuzz testing projects (such as in [9], [17]) aim to randomly fuzz certain data values to identify their value constraints or to explore the program space.

TABLE 6
Value Mutation Rules

Type	Mutation Rules
ID	0, copy from another ID, increment or decrement by a small constant
Size/Offset	0, increment or decrement by a small constant
String	“”, copy from another string, mutate one character

TABLE 7
Semantic Fields Selected for Windows XP SP3 and Their Mutability

Category: Structures	Semantic Field	Mutability
Process: struct EPROCESS	UniqueProcessId, ExitStatus, ImageFileName, CreateTime, GrantedAccess, InheritedFromUniqueProcessId, ObjectTable.HandleCount, ObjectHeader.ObjectType	✓
	ActiveThreads, Flags	✗
	Token	p
Thread: struct ETHREAD	StartAddress, Cid.UniqueThread, ObjectHeader.ObjectType	✓
	Cid.UniqueProcess	✗
DLL & Kernel Module: struct LDR_DATA_TABLE_ENTRY	DllBase, EntryPoint, FullDllName, BaseDllName, Flags, LoadCount, PatchInfo	✓
Registry Key, CM_KEY_NODE	Name, NameLength, LastWriteTime, SubkeyCounts, Flags, Signature, Parent	✓
	Security	✗
Network	TCPT_OBJECT.RemoteIpAddress, TCPT_OBJECT.RemotePort, TCPT_OBJECT.LocalAddress, TCPT_OBJECT.LocalPort, TCPT_OBJECT.Pid	✗
	TCP_LISTENER.AddressFamily, TCP_LISTENER.Owner, TCP_LISTENER.CreateTime, TCP_ENDPOINT.State	✓
Memory Pool: struct POOL_HEADER	PoolTag, BlockSize	✓

We list the mutation rules in Table 6. For example, for an ID (e.g., pid, tid), we consider 0 as an input, because 0 is often reserved for system process and thread. Similarly, for a string, we use an empty string as an input since the OS may have special handling for empty strings, such as ignoring and skipping an object if its name is empty. An attacker may exploit this feature to hide certain objects.

4.2 Experiment Setup

We perform our empirical study on two popular operating systems, which are Windows XP with service pack 3 (XPSP3) and Ubuntu 10.04 with Linux kernel version 2.6.32-25 (Linux). We conducted the experiments on a system with Intel Pentium Core i7 3 GHz processor and 4 GB RAM. The host operating system is 32-bit Ubuntu 10.04 with kernel version 2.6.32-38. We analyzed both operating systems individually as a virtual machine running inside DECAF. 512 MB RAM was allocated for the virtual machine.

We compiled two lists of semantic fields, one for Windows XP (Table 7) and the other for Linux (Table 8). Forensic tools (such as Volatility [2]) query these semantic fields to extract semantic information from a memory dump. Although these lists are not nearly complete, we believe that they provide a fairly good coverage on important semantic fields.

Using the value mutation rules listed in Table 6, we designed three mutation tests (including one whole-set mutation) for each field in Tables 7 and 8 resulting in a total of 258 test cases. The test cases were distributed across 12 test points (six test points in each of the two OSs), with

TABLE 8
Semantic Fields Selected for Linux and Their Mutability

Category: Structures	**Semantic Field**	**Mutability**
Process: struct task_struct	state, flags, comm, start_time, stime, exit_code	✓
	fds	✗
	pid	*p*
File: struct dentry struct inode	task_struct.files.fd[i].f_owner, task_struct.files.fd[i].f_mode, task_struct.files.fd[i].f_pos, d_name, d_iname, d_flags, d_time, i_uid, i_gid, i_size, i_atime, i_ctime, i_mtime	✓
Module: struct module, struct vm_area_start	name, num_syms, state, core_size, core_text_size, num_kp, vm_flags	✓
	vm_start, vm_end,	✗
Network: struct inet_sock, struct sock_common, struct sock	saddr, daddr, sport, dport	✗
	skc_family, skc_refcount, skc_state, sk_protocol, sk_flags, sk_type, sk_err	✓

average trace gathering time of approximately 15 minutes per test point. Depending on the test point in question and the size of trace, redundancy identification and semantic value location took between 7 min (best case) to 32 min (worst case) with 92 percent of the time consumed during redundancy identification. Each test case execution involving VM restoration and fuzzing took 25 to 60 seconds. After fuzzing, the execution continued for 3 minutes as a part of behavior assessment. Additionally, we implemented a rootkit to examine the effects of semantic mutations on the OS information retrieval tools. In one shot, we mutated the primitives listed in Table 11 and observed the impact on the system.

The key component of MOSS is *Duplicate Semantic Value Analysis*, which in theory is independent of the OS. Therefore, with the kernel data structure information for the key kernel data structures, careful identification of test points and a corresponding test program, one can perform single-field and duplicate-field mutations on any guest OS to identify the semantic fields susceptible to mutation. In this paper, as a proof-of-concept, we consider Windows XP SP3 and Linux 2.6.32-25 to perform the empirical study. However, it is often the case that a new version of an OS retains a significant part of the previous version. Therefore, it is possible that the mutability results tabulated in Tables 7 and 8 are applicable to other versions of Windows and Linux OSes, respectively.

4.3 Single Field Mutation

We consider a semantic field to be immutable only if all of the mutation attempts on it cause system or program instability. If some of the mutations do not cause critical failures, then attackers may potentially make similar modifications and thus mislead the security tools. Based on this standard, we have listed the results in the last column of Tables 7 and 8. A 'p' in the mutability column indicates that the semantic field showed no system or program instabilities for certain mutations, while it did for some others.

From the mutability column in Tables 7 and 8, we can see that most of the semantic fields, including process name, file name, module name and many others can be changed by an attacker without adverse effects on the system or a program. This observation immediately raises a question about the trust issue for all the security applications (such as memory forensics and virtual machine introspection) that critically rely on the correctness of these semantic fields.

For both operating systems, network related semantic fields tend to be reliable. Mutations to source and destination IP addresses and port numbers immediately cause failures to subsequent operations on the network connection. This is good news, implying that network security tools that make security decisions based on the network connections can be trusted, as long as these connection objects can be reliably located.

For Windows XP, the UniqueProcessId in ETHREAD tends to be reliable. A mutation will either crash the entire system or the test program. The Pid in the TCP connection object (TCPT_OBJECT) can also be relied upon. A mutation on it will immediately drop the connection. It is worth noting that security tools usually read Pid from EPROCESS.UniqueProcessId, which turns out to be not reliable at all, because none of the mutations on it causes severe failures. This finding suggests to retrieve the UniqueProcessId in the ETHREAD objects or Pid in the TCPT_OBJECT objects (if available) instead.

Interestingly, strings are completely mutable (that is, all occurrences of the string can be mutated without adverse effect on the system) for both the operating systems we tested. OS kernels usually rely on pointers and integers (such as handles and IDs) for operations as opposed to strings. String mappings for resources (e.g., file handle to file name) are often maintained in instances that involve interpretation by a human. This observation is particularly worrisome since strings like process name, file name, registry key name, etc., have severe security relevance and are fully mutable.

Similarly, it turns out that all the time related information (such as, process creation time, exit time, etc.) are also fully mutable and therefore not reliable. This observation has far reaching impacts. For instance, time information is crucial in a memory forensic context. One may need to use the time stamps of certain malicious activites as crime evidence. With DKOM as a possibility, such time stamps cannot be assumed correct.

4.4 Duplicate Field Mutation

In addition to mutating the selected semantic fields individually, we also identified their duplicate fields and mutated these duplicates both separately and simultaneously. We present these results in Tables 9 and 10 for Windows XP and Linux respectively. For each primary semantic field that has at least one duplicate, we list the number of duplicates (including the primary) identified through MOSS, the types of these duplicates, immutable duplicates if any, and whether the entire duplicate set is mutable. Due to the dynamic nature of our analysis, the number of duplicates depends on the start execution point, the end execution point, and the particular execution path. In our experiment,

TABLE 9
Duplicate Fields for Windows XP and Their Mutability

Primary Field	# of Dups	Type of Duplicates	Immutable Duplicates	Set Mutability
_EPROCESS.UniqueProcessId	36	_ETHREAD.Cid.UniqueProcess, _HANDLE_TABLE.UniqueProcessId, _CM_KEY_BODY.ProcessId, _EPROCESS.InheritedFromUniqueProcessId, _ETIMER.Lock, _TEB.ClientId, _TEB.RealClientId, 0x9b57b6d0, 0x9ccdaef0, 0x9cce697c...	_ETHREAD.Cid.UniqueProcess	✗
_EPROCESS.ImageFileName	4	_OBJECT_NAME_INFORMATION.Name, _RTL_USER_PROC_PARAMS.ImagePathName, _SE_AUDIT_PROCESS_INFO.ImageFileName	None	✓
_EPROCESS.CreateTime	2	_ETHREAD.CreateTime	None	✓
_EPROCESS.ActiveThreads	2	_EPROCESS.ActiveThreadsHighWatermark	None	✓
_HANDLE_TABLE.HandleCount	2	_HANDLE_TABLE.HandleCountHighWatermark	None	✓
_FILE_OBJECT.FileName (Data file)	7	0x003a948e, 0x822df33a, 0x822df35c, ...	None	✓
_LDR_DATA_TABLE_ENTRY.FullDllName	3	_LDR_DATA_TABLE_ENTRY.BaseDllName, _FILE_OBJECT.FileName	None	✓
_LDR_DATA_TABLE_ENTRY.BaseDllName	3	_LDR_DATA_TABLE_ENTRY.FullDllName, _FILE_OBJECT.FileName	None	✓
_CM_KEY_NODE.LastWriteTime	2	0x9b43ea60	None	✓
_CM_KEY_NODE.Parent	4	0x94d20a20, 0x9adc7940, 0x9adc7948	None	✓
_CM_KEY_NODE.Security	2	0x822c7880	_CM_KEY_NODE.Security	✗
_ETHREAD.StartAddress	2	_SECTION_OBJECT.StartingVa	_SECTION_OBJECT.StartingVa	✗

duplicate values were identified by dynamic duplicate value analysis from the start of the test program to a predetermined test point. Therefore, these duplicates may not always hold true for different test cases. For each duplicate value, we further identify the data structure and field in which the value is located. Again, we use Volatility for locating kernel data structures. Due to the limited coverage of Volatility, we may not always be able to recognize the corresponding data structures. In such cases, we list only the virtual addresses in the third column.

The immutable duplicates, if any, indicate which duplicate fields (other than the primary) *may* be reliable. The knowledge about immutable duplicates is valuable, because it means that security tools could examine these alternative fields instead of the primary ones to obtain more reliable OS semantics.

The last column indicates if the entire duplicate set is simultaneously mutable. If not, security tools may be able to perform a consistency check on the entire set to obtain more reliable outputs. Of course, the underlying assumption is that the security tool is smart enough to locate all the duplicate fields, which in practice may be difficult, especially for closed-source operating systems like Windows.

From the results in Tables 9 and 10, we can see that information redundancy does exist for some important OS semantics. This is the case for both operating systems. For example, in Windows, EPROCESS.UniqueProcessId appears as the UniqueProcess in all the ETHREAD objects belonging to that process, and also appears in the HANDLE_TABLE. For a process which has established at least one TCP connection, the pid should also appear in the TCPT_OBJECT.pid [18], which MOSS could not identify at test point 1. This is because the network operations happened after test point 1 in our experiment and the corresponding TCPT_OBJECT was not created at that point. In fact, at test point 5, we confirmed that TCPT_OBJECT.pid indeed is one of the duplicates. For the process name – EPROCESS.ImageFileName, we found duplicates in OBJECT_NAME_INFORMATION.Name and RTL_USER_PROCESS_PARAMETERS.ImagePathName. As the main module, the process name also appears in the base module name BaseDllName and full module name FullDllName in LDR_DATA_TABLE_ENTRY. These results are also consistent with publicly available Windows documentation [18].

For Linux, we found that the pid of the test program replicates in the group id task_struct.t_gid, and also the light-weight process (lwp)'s group id, which specifies the pid of the hosting process of a thread in Linux. Similarly, the process name in task_struct.comm also shares the same value with its light-wight processes. vma.vm_start has a duplicate in vma.vm_end of the preceding vma structure, and vma.vm_end has a duplicate in vma.vm_start of the subsequent vma structure. We also found that the source IP address and the destination IP address are duplicate to each other. This is because in our test, both the server and the client programs are running in the localhost, so both source and destination IP addresses are 127.0.0.1. These findings are in agreement with the source code of the OS kernel.

Unfortunately, our results show that most of these duplicate values are mutable both individually and simultaneously. In very limited cases, the information redundancy can help improve the integrity of semantic information. As discussed earlier, though UniqueProcessId in EPROCESS is mutable, its duplicate, UniqueProcess in ETHREAD is immutable. ETHREAD.StartAddress in Windows is another such case. The primary ETHREAD.StartAddress can be manipulated, but its duplicate StartingVa in _SECTION_OBJECT is more sensitive to mutations.

Tables 9 and 10 also show that the result of mutating the entire duplicate set is the same as mutating the individual duplicate fields. This indicates that the operating systems process these semantic fields separately, and perform no cross checking on these duplicates. From the defender's perspective, if one can reliably locate one immutable field

TABLE 10
Duplicate Fields for Linux and Their Mutability

Primary Field	# of Dups	Type of Duplicates	Immutable Duplicates	Set Mutability
task_struct.pid	4	task_struct.t_gid, task_struct.t_gid(lwp), 0xf63916dc	None	✓
task_struct.comm	2	task_struct.comm(lwp)	None	✓
task_struct.static_prio	3	task.parent.static_prio, task.static_prio (lwp)	None	✓
task_struct.exit_code	3	task.parent.exit_code, task.exit_code (lwp)	None	✓
task_struct.fds	3	0xf7179080, 0xf61bae84	0xf7179080, task.fds	✘
module.name	2	0xd93c524c	None	✓
module.num_syms	12	module.num_kp, 0xe086c15c, 0xe086c170...	None	✓
vma.vm_start	2	vma.vm_end	vma.vm_start	✘
vma.vm_end	2	vma.vm_start	vma.vm_end	✘
dentry.d_name	2	0xf583f0d8	None	✓
inet_sock.saddr	24	inet_sock.rcv_saddr inet_sock.daddr 0xde49147c 0xde49148c . . .	inet_sock.rcv_saddr inet_sock.daddr 0xde49147c 0xde49148c . . .	✘
inet_sock.daddr	24	inet_sock.rcv_saddr inet_sock.saddr 0xde49147c 0xde49148c . . .	inet_sock.rcv_saddr inet_sock.daddr 0xde49147c 0xde49148c . . .	✘

(either the primary or a duplicate), checking the entire duplicate set is not necessary.

4.5 Impact on Security Tools

System administration and security tools are a doorway for system administrators to investigate various aspects of a running OS. If an SVM attack can either successfully misrepresent (i.e., provide false information) or completely evade (i.e., hide from) security tools, it impacts investigation. To this end, we selected a set of adminstration and security tools. Specifically, we picked (1) Task Manager and Process Explorer [18] that use Windows APIs to retrieve information from the OS, (2) Volatility plugins (scan based and traversal based) that directly operate on the guest OS memory, (3) SigGraph [10][2] and Robust Signatures—two of the state-of-the-art tools to detect data structures in the memory and finally (4) DECAF [14], an in-house state-of-the-art execution monitor built on top of QEMU. DECAF retrieves semantic information from a guest operating system by first locating the global data structures in the guest memory and traversing them. Specifically, it can retrieve process, thread and module information from the guest operating system. These tools were selected based on free availability, popularity and technique used to retrieve information from the guest OS.

To evaluate the impact on the above tools and to further emphasize the results from fuzz testing in Tables 9 and 10, we implemented a proof-of-concept SVM rootkit for Windows. Given a specific malware process, this SVM rootkit manipulates all the mutable semantic fields associated with the process, and their duplicates that can be identified. The SVM rootkit changes the integer values to 0 and string values to empty string. Also, the rootkit changes the pool tags to "None" to indicate that the object is associated with the default pool. To demonstrate the power of this rootkit, we ran a bot named TDSS [12] in Windows XP SP3 in a controlled environment. We kept TDSS running for over 3 hours before recording the outputs of various security tools, to ensure that neither the system nor the program crashed because of the SVM rootkit.

Table 11 lists the impact of the SVM rootkit on the selected security tools. It presents what primary semantic fields are manipulated and the mutation impacts on the security tools. We can see that there are mainly two kinds of symptoms: either the OS entities become hidden (H), or the misleading new values are fetched and displayed (N). Volatility traversal tools (i.e., `pslist`, `threads`, `modules`) are misled to show meaningless values. For scan tools, while the process and thread information is hidden from `psscan` and `thrdscan`, `modscan` can still identify the module information, which unfortunately has been manipulated and thus has become meaningless. The reason why process and thread objects are hidden is because their pool tags have been manipulated, and `psscan` and `thrdscan` rely on pool tags to identify process and thread objects. The two robust signature schemes are also misled or evaded. The graph signature [10] for `EPROCESS` is reliable enough to find malware's process, but the obtained process information is invalid. The value-invariant signature [9] is even worse. It failed to identify the malware process because the `ExitTime` of the malware process has been manipulated and the signature uses the `ExitTime` value to remove noisy and dead process objects. The result of DECAF's VMI tool is similar to that of the Volatility traversal tools. That is, although the information about the malware execution can be extracted, it is incorrect. Consequently, unless the integrity of the kernel can be guaranteed, we cannot leverage the knowledge obtained from VMI tools to perform analysis on the malware execution.

Root-Causes. Identifying the true causes why SVM attacks succeed on a closed source OS like Windows is hard. We found three main causes that make SVM attacks possible.

First, by design, there is an inherent presumption of data integrity within the kernel. From the perspective of security applications, the information retrieved from the kernel is assumed correct and corner cases like IDs being "NULL", names being empty, etc. may not be handled. For example, if the process name is an empty string, one application may display the remaining process attributes with an empty name whereas, another application may completely skip

[2] We implemented SigGraph as a plugin to Volatility and created a signature for EPROCESS.

TABLE 11
Impact of SVM Rootkit on Security Tools

Category	Primary Fields Mutated	Task Mgr	Proc Exp	Volatility (scan)	Volatility (traversal)	Sig Graph	Robust Sign	VMI
Process	EPROCESS.UniqueProcessId	H	H	H	N	N	H	N
	EPROCESS.InheritedFromUniqueProcessId	-	H	H	N	N	H	N
	EPROCESS.POOL_HEADER.PoolTag	-	-	H	N	N	H	N
	EPROCESS.POOL_HEADER.BlockSize	-	-	H	N	N	H	N
	EPROCESS.CreateTime	-	H	H	N	N	H	N
	EPROCESS.ExitTime	-	H	H	N	N	H	N
	EPROCESS.ImageFileName	H	H	H	N	N	H	N
	EPROCESS.ExitStatus	-	-	H	N	N	H	N
Thread	ETHREAD.CreateTime	-	H	H	N	-	-	N
	ETHREAD.ExitTime	-	H	H	N	-	-	N
	ETHREAD.Cid.UniqueThread	-	H	H	N	-	-	N
	ETHREAD.StartAddress	-	H	H	N	-	-	N
	ETHREAD.POOL_HEADER.PoolTag	-	H	H	N	-	-	N
	ETHREAD.POOL_HEADER.ObjectSize	-	H	H	N	-	-	N
Kernel Module & User DLL	LDR_DATA_TABLE_ENTRY.DllBase	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.EntryPoint	-	-	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.SizeOfImage	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.FullDllName	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.BaseDllName	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.Flags	-	-	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.PatchInformation	-	H	N	N	-	-	N
	LDR_DATA_TABLE_ENTRY.LoadCount	-	H	N	N	-	-	N

"H" indicates that the entity was completely hidden from the tool when the value was set to 0 or an empty string, and a value of "N" indicates that the the tool reports the mutated value.

the process as invalid. Depending on the application behavior, an attacker can pick the mutations that will have the desired effect.

Second, kernel functionality seldom depends on strings and certain time related fields, and mutations to such fields have no effect on the system. This is expected since the kernel mainly uses strings for bookkeeping and user interaction. For example, in the linux kernel, the process name is represented by the `task_struct.comm` field in "sched.h". Its usage is mainly confined to logging and error reporting.

Finally, we found that the kernel APIs do not take mutability into consideration while retrieving semantic values because kernel code tends to prioritize performance and convenience over mutability. This is reasonable since direct mutations to data is not an attack vector in the OS threat model. For example, in `GetProcessId` API, instead of retrieving the process ID from an immutable semantic (if any) such as a process' thread (i.e., `ETHREAD.Cid.UniqueProcess`), the process ID is retrieved from `EPROCESS.UniqueProcessId`, which is mutable.

4.6 Attack Severity

An attack is severe if through a successful SVM attack, an attacker can mislead an administrator to accomplish malicious tasks. Note that—from Sections 4.1 and 4.2—an attack is successful if the mutation does not result in any adverse affect on the system. From Table 11, we see that mutating certain fields can hide entities or mis-represent their values to the examining tools. Administrators often examine the modules running within a system and schedule forensic tasks if an unrecognized (or non-whitelisted) module is detected. Through SVM attacks, for example, an attacker could change the load address and name of a malicious DLL to impersonate a well known non-malicious DLL (such as "kernel23.dll") thereby evading further examination. In fact, using such a technique, we were able to successfully mis-represent a malicious DLL when the memory was examined using Volatility.

Moreover, in both OSs, changing the name of the file in the file object and all the duplicates, the file becomes completely inaccessible even to an administrator. An attacker could hide in plain sight using such a technique. An anti-virus or other state-of-the-art forensic tools can neither delete nor quarantine such files. Also, in Windows, by changing the `EPROCESS.Flags` value to $0xFFFFFFFF$, an attacker could prevent the process from termination. Upon attempting to terminate the process, a dialog pops up with a message that the process is a system process and terminating it would result in a restart. Force quitting it abruptly restarts the OS. An attacker may use this trick to prevent a malicious process from being killed.

While some of the tasks like hiding a process, escalating privileges, etc. can be accomplished by modern rootkits, they usually involve complex operations requiring extensive domain knowledge. Our observations highlight how even simple mutations to semantic fields could yield drastic outcomes. Moreover, to the best of our knowledge, no existing rootkit considers *all* duplicate semantic values during DKOM.

5 Discussions and Future Work

Memory analysis on closed-source OSs lacks completeness. As demonstrated by Volatility, while community's domain knowledge may be sufficient to harness heuristics to gather information from a memory image, it can not be deemed complete. False positives can be analyzed and excluded by a domain expert, but there is no way to know about the existence of a missed object. We have highlighted the usefulness of ground-truth by relying on it to identify a bug in the `filescan` plugin of Volatility. In a deceptive context, we believe reliance on ground-truth is a requirement.

SVM attack space is vast. In our experiments, we have limited our changes to single value mutations and duplicate set mutations, but an attacker is not restricted to making these changes. In general, we did not attempt to test multiple mutations, since it leads to a very large number of combinations, which are infeasible to test. However, our current testing infrastructure does support multi-mutation based tests and can be extended in future. The key focus of our tests are to highlight the seriousness of single value and duplicate set mutations, which we believe is a large attack space in itself.

We need more trustworthy VMI techniques. The current VMI techniques [3], [5], [19], [20] more or less rely on memory analysis, and can therefore be incorrect. Triggered by certain events (e.g., system calls) or demanded by the administrator, the current VMI techniques traverse important kernel data structures of the guest system, and then extract the operating system semantics. Virtuoso [19] and VMST [20] have greatly narrowed the semantic gap and improved the usability of VMI, but these new approaches do not change the fact that they directly read from the virtual machine memory, disregard of other runtime events. Once the guest kernel is compromised, the current VMI techniques will fail, just like memory forensics.

6 Related Work

Dynamic Binary Analysis. Dynamic taint analysis has been extensively used to solve various security problems, such as data life time tracking [21], exploit detection [15], vulnerability fuzzing [17], malware analysis [4], protocol reverse engineering [22], [23], [24], and data structure reverse engineering [25], [26], [27]. We leverage dynamic taint analysis to construct the ground-truth and reliably identify links between kernel objects.

Our approach for data structure reconstruction is also different from the previous approaches, such as REWARDS [25] and Howard [27]. As we are mainly interested in the inter-connections between kernel objects, we only monitor the lifetime of kernel objects and keep track of pointers. With this trade-off, dynamic analysis can be efficient enough to analyze the kernel execution. In comparison, REWARDS and Howard perform more heavyweight instrumentation on each instruction, and thus can not be directly applied to analyze the kernel execution in a timely manner. At some level, our algorithm to track equivalent variables and direct our mutation is related to the abstract variable binding technique for automatic reverse engineering of malware emulators [28], in which two kinds of dataflow algorithms (i.e., forward binding and backward binding) are proposed.

Kernel rootkit detection. Rootkit is a kernel-level malware, which hides the presence of important kernel objects. It has posed a significant threat to the integrity of operating systems. Early research uses specification based approach deployed in hardware (e.g., [29]), or binary analysis [30] to detect kernel rootkits. Recent advances include state-based control flow integrity checking (e.g., SBCFI [31] and KOP [32]), and data structure invariant based checking [9], [10], [33]. Patagonix [34] maintains a database of approved binaries and taps into hardware features to detect all running executables and thereby detect unapproved binaries. Our work extends rootkit DKOM techniques by exploring automated single- and redundant-value manipulation based attacks.

Virtual machine introspection. Introspecting a virtual machine often requires interpreting the low level bits and bytes of guest OS kernel to high level semantic state. This is a non-trivial task, because of the semantic-gap [7]. Early approaches [3], [29], [35] have used manual efforts to locate the kernel objects by traversing from the exported kernel symbols or searching for invariants. Recent advances show that we can largely automate this process [19]. Prakash et al. [36] leverage hardware events to recover key semantics of interests. Our work sheds some light on the VMI techniques. We show that in many cases the semantic knowledge extracted by VMI cannot be trusted, and we call for more trustworthy VMI techniques.

7 Conclusion

We have conducted an empirical study on the state-of-the-art memory analysis tools, especially those for Windows—a closed-source operating system. To acquire the ground truth, we have devised dynamic binary analysis techniques. Our experimental results demonstrate that both traversal-based and signature-based analysis tools are not perfectly accurate even under a non-deceptive context. To further evaluate the attack space of value manipulation attacks, we designed a value equivalence set directed field mutation technique. Through our directed mutation, we found that these attacks are practical and many semantic values can be altered without being noticed and cause adverse effect to the victim system.

Acknowledgments

The authors would like to thank anonymous reviewers for their comments. This research was supported in part by NSF grant #1018217, NSF grant #1054605, AFOSR grant #FA9550-12-1-0077 and #FA9550-14-1-0119, McAfee Inc, and VMware Inc. Any opinion, findings, conclusions, or recommendations are those of the authors and not necessarily of the funding agencies.

References

[1] N. L. Petroni, Jr, A. Walters, T. Fraser, and W. A. Arbaugh, “FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory,” *Digital Investigation*, vol. 3, no. 4, pp. 197–210, 2006.

[2] Volatility: Memory Forensics Framework. (2014). [Online]. Available: https://code.google.com/p/volatility/

[3] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through VMM-based ‘out-of-the-box’ semantic view reconstruction,” in *Proc. 14th ACM Conf. Comput. Commun. Security*, Oct. 2007, pp. 128–138.

[4] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis,” in *Proc. 14th ACM Conf. Comput. Commun. Security*, Oct. 2007, pp. 116–127.

[5] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Netw. Distrib. Syst. Security Symp.*, Feb. 2003, pp. 191–206.

[6] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: An efficient 'out-of-vm' approach for fine-grained process execution monitoring," in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 363–374.

[7] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. 8th Workshop Hot Topics Oper. Syst.*, 2001, pp. 133–138.

[8] (2005). FU Rootkit [Online]. Available: http://www.rootkit.com/project.php?id=12

[9] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proc. ACM Conf. Comput. Commun. Security*, 2009, pp. 566–577.

[10] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," presented at the Proc. 6th ACM Symp. Inf., Comput. Commun. Security, San Diego, CA, USA, 2011.

[11] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh, "Openconflict: Preventing real time map hacks in online games," in *Proc. IEEE Symp. Security Privacy*, 2011, pp. 506–520.

[12] S. Golovanov. (Aug., 2010). Analysis of TDSS rootkit technologies. [Online]. Available: http://securelist.com/analysis/36314/tdss/

[13] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, "Manipulating semantic values in kernel data structures: Attack assessments and implications," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2013, pp. 1–12.

[14] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in *Proc. 2014 Int. Symp. Softw. Test. Anal.*, 2014, pp. 248–258.

[15] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. 12th Annu. Netw. Distrib. Syst. Security Symp.*, 2005, pp. 196–206.

[16] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Design Implementation*, 1990, pp. 246–256.

[17] P. Godfroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. 15th Annu. Netw. Distrib. Syst. Security Symp.*, Feb. 2008, pp. 151–166.

[18] M. Russinovich. Windows sysinternals utilities. (2014). [Online]. Available: http://technet.microsoft.com/en-us/sysinternals

[19] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. IEEE Symp. Security Privacy*, May 2011, pp. 297–312.

[20] Y. Fu and Z. Lin, "Space traveling across VM: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection," in *Proc. 2012 IEEE Symp. Security Privacy*, pp. 586–600.

[21] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proc. 13th USENIX Security Symp.*, 2004, pp. 586–600.

[22] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 317–329.

[23] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," in *Proc. 15th Annu. Netw. Distrib. Syst. Security Symp.*, 2008, pp. 1–14.

[24] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proc. 15th Annu. Netw. Distrib. Syst. Security Symp.*, 2008, vol. 8, pp. 1–15.

[25] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proc. 17th Annu. Netw. Distrib. Syst. Security Symp.*, 2010, vol. 10, pp. 1–18.

[26] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2011, pp. 251–268.

[27] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," presented at the Proc. Netw. Distrib. Syst. Security Symp., San Diego, CA, USA, 2011.

[28] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proc. 30th IEEE Symp. Security Privacy*, 2009, pp. 94–109.

[29] N. L. Petroni, Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - A coprocessor-based kernel runtime integrity monitor," in *Proc. 13th USENIX Security Symp.*, 2004, p. 13.

[30] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *Proc. 20th Annu. Comput. Security Appl. Conf.*, 2004, pp. 91–100.

[31] J. Nick, L. Petroni, and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 103–115.

[32] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proc. ACM Conf. Comput. Commun. Security*, 2009, pp. 555–565.

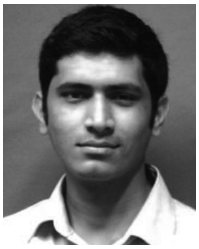
[33] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *Proc. Annu. Comput. Security Appl. Conf.*, 2008, pp. 77–86.

[34] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proc. 17th Usenix Security Symp.*, Jul. 2008, pp. 243–258.

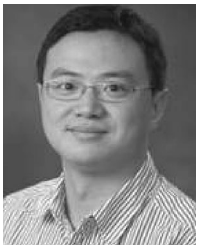
[35] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. 11th Int. Symp. Recent Adv. Intrusion Detection*, 2008, pp. 1–20.

[36] A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *Proc. 8th ACM SIGSAC Symp. Inf., Comput. Commun. Security*, 2013, pp. 311–322.

Aravind Prakash is a fifth year PhD student at Syracuse University. He works on system security with focus on program analysis, memory forensics, exploit diagnosis, and mobile security.



Eknath Venkataramani received the master's degree in computer science from Syracuse University. After graduating, he has been with McAfee Labs as an anti-malware researcher. His research interests include rootkits, system security, reverse engineering, and malware analysis.



Heng Yin received the PhD degree in Computer Science from the College of William and Mary in 2009. He is an assistant professor in the Department of Electrical Engineering and Computer Science at Syracuse University, Syracuse, New York. His research interests lie in binary code analysis, mobile system security, virtualization, etc. He is a member of the IEEE.

Zhiqiang Lin received the PhD degree from Purdue University in 2011. He is an assistant professor with the Computer Science Department, University of Texas at Dallas. His current research focuses on system and software security with an emphasis on binary code reverse engineering, vulnerability discovery, malicious code analysis, and OS kernel protection. He is a member of the IEEE.