DIGGER: Identifying OS Kernel Objects for Run-time Security Analysis

Amani S. Ibrahim, James Hamlyn-Harris, John Grundy and Mohamed Almorsy

Centre for Computing and Engineering Software Systems
Swinburne University of Technology
Melbourne, Australia
[aibrahim, jhamlynharris, jgrundy, malmorsy]@swin.edu.au

Abstract — **In operating systems, we usually refer to a running instance of a data structure (data type) as an *object*. Locating dynamic runtime kernel objects in physical memory is the most difficult step towards enabling implementation of robust operating system security solutions. In this paper, we address the problem of systemically uncovering all operating system dynamic kernel runtime objects, without any prior knowledge of the operating system kernel data layout in memory. We present a new hybrid approach – called DIGGER – that uncovers kernel runtime objects with nearly complete coverage, high accuracy and robust results. The information revealed allows detection of generic pointer exploits and data hooks. We have implemented a prototype of DIGGER and conducted an evaluation of its efficiency and effectiveness. To demonstrate our approach's potential, we have also developed three different proof-of-concept operating system security tools based on the DIGGER approach.**

Keywords - Operating Systems, Kernel Data Structures, Dynamic Runtime Objects.

I. INTRODUCTION

Dynamic kernel runtime objects can be a significant source of security and reliability problems in Operating Systems (OSes). Locating those objects from a trusted source is an important step towards enabling implementing robust operating system security solutions. The problem with dynamic kernel runtime objects is the complex and unpredictable runtime memory layout of those objects. An operating system kernel has thousands of heterogeneous data structures that have direct and indirect relations between each other with no explicit integrity constraints, providing a large attack surface to hackers. In Windows and Linux operating systems (from our analysis) nearly 40% of the inter-data structure relations are pointer-based relations (indirect relations), and 35% of these pointer-based relations are generic pointers (i.e. null pointers that do not have values, and void pointers that do not have associated type declarations in the source code) [20, 21]. Such generic pointers get their values or type definitions only at runtime according to the different calling contexts in which they are used [1]. In such a complex data layout, the runtime memory layout of the data structures cannot be predicted at compilation time. This makes the kernel data a rich target for rootkits that exploit the points-to relations between data structure instances in order to hide or modify system runtime objects.

Accurately identifying the running instances of the OS kernel data structures and objects is an important task in many OS security solutions such as kernel data integrity checking [2, 3], memory forensics [4, 5], brute-force scanning [6], virtualization-aware security solutions [7, 8], and anti-malware tools [9, 10]. Although discovering runtime objects has been an aim of many OS security research efforts, existing and proposed solutions still have a number of limitations. Most fall into two main categories: *Memory Mapping Techniques* and *Value-Invariant Approach*.

Memory Mapping Techniques. Memory mapping techniques such as CloudSec [7], KOP [11] and OSck [3] identify kernel runtime objects by recursively traversing the kernel address space starting from the operating system global variables and then follow pointer dereferencing until reaching the running object instances, according to a predefined kernel data definition. This kernel data definition reflects the runtime kernel data layout in memory, and is specific for each kernel build. However, memory traversal techniques are limited and not very accurate, because:

- They are vulnerable to kernel data rootkits that exploit the points-to relations between data structures running instances, which hide the runtime objects or point to somewhere else in the kernel address space.
- They require a predefined definition of the kernel data layout that accurately disambiguates indirect points-to relations between data structures, in order to enable accurate mapping of memory. However – to the best of our knowledge – all of the current efforts (with the exception of KOP [11]) depend on a security expert's knowledge of the kernel data layout to manually resolve ambiguous points-to relations. Thus, those approaches only cover 28% of kernel data structures (as discussed by Carbone *et al.* [11]) that relate to well-known objects such as processes, threads and loaded modules.
- They are not effective when memory mapping and object reachability information are not available. Sometimes security experts need to make a high-level interpretation of a set of memory pages where the mapping information is not available *e.g.* in system crash dumps. Incomplete subsets of memory pages cannot be traversed, and thus data that resides in the absent pages cannot be recovered.
- They have a high performance overhead because of poor spatial locality. The problem with general-purpose OS allocators is that objects of the same type could be scattered around the memory address space. Traversal of the physical memory therefore requires accessing several memory pages.
- Finally, they (with the exception of KOP [11]) cannot follow generic pointer dereferencing as they only

leverage type definitions, thus cannot know the target types of these untyped pointers.

Value Invariants Approaches. Value-invariants approaches such as DeepScanner [10], DIMSUM [12] and SigGraph [6], use the value-invariants of certain fields or of a whole data structure as a signature to scan the memory for matching running instances. However, such a signature may not always exist for a data structure, as discussed by Lin *et al.* [6]. Moreover, many kernel data structures cannot be identified by such value-invariant schemes. For example, it is difficult to generate value-invariants for data structures that are part of linked lists (single, doubly or triply), because the actual running contents of those structures depends on the calling contexts at runtime. In addition, the value-invariant approach does not fully exploit the range of generic pointers in data structures fields, and is not able to uncover the points-to relations between the different data structures. Finally, the performance overhead of those approaches is extremely high, as they scan the whole kernel address space with large signatures, as they typically include most data structure fields in the signature.

Motivated by the limitations of the current approaches and the need to accurately identify the runtime dynamic kernel objects from a robust view that cannot be tampered with, we have developed a new approach called DIGGER. DIGGER is capable of systematically uncovering all system runtime objects without any prior knowledge of the operating system kernel data layout in memory. Unlike previous approaches, DIGGER is designed to address the challenges of indirect points-to relations between kernel data structures. DIGGER employs a hybrid approach that combines new value-invariant and memory mapping approaches in order to get accurate results with nearly complete coverage. The value-invariant approach is used to discover kernel object instances with no need for memory mapping information. The memory mapping approach is used to retrieve the object's details in depth including *points-to* relations (direct and indirect) with the other running data structures, without any prior knowledge of the operating system kernel data layout.

DIGGER first performs offline static *points-to* analysis on the kernel's source code to construct a type-graph that summarizes the different data types located in the operating system kernel along with their connectivity patterns, and the candidate target types and values of generic pointers scattered around the kernel. The type-graph is not used to discover running object instances. It is used to enable systematic memory traversal of the object details with no need for the symbol information or operating system expert knowledge. Second, DIGGER uses the four-byte pool memory tagging schema as a new value-invariant signature – that is not related the data structure layout – to uncover kernel runtime dynamic objects instances from the kernel address space.

DIGGER's approach has accurate results, a low performance overhead, fast and nearly complete coverage, and zero rates of false alarms. We have implemented a prototype system using DIGGER and evaluated it on the Windows OS to prove its efficiency in discovering: *(i)* kernel runtime objects; *(ii)* terminated objects that still persist in the physical memory; and *(iii)* semantic data of interest in dead memory pages. To demonstrate the power of DIGGER, we have also developed and evaluated three OS security prototype tools based on it, namely, B-Force, CloudSec+ and D-Hide. B-Force is a brute force scanning tool. D-Hide is a tool that can systematically detect any hidden kernel object type. It is not just limited to the well-known objects. CloudSec+, a virtual machine (VM) monitoring tool, is used in virtualization-aware security solutions to externally monitor and protect a VM's kernel data.

Section II gives an overview on the operating system kernel data problem and key related work. Section III presents our DIGGER approach, and section IV explores its implementation and evaluation. In section V, we explore the operating system security tool prototypes and Finally we discuss results and draw key conclusions.

II. Background

Data hooks have the ability to modify the running instances of kernel runtime objects without injecting any malicious code, in order to hide running objects (*e.g.* processes, loaded modules, drivers or services) or alter operating system behavior. In OSes we usually refer to a running instance of a data structure as an *object*. Data structures scatted around operating system kernels can be classified into two categories:

- *Control Data Structures (CDS)*; these are the static data structures that are used in control-transfer instructions such as system call tables and descriptor tables. These data structures do not change during runtime in value, location or number of the running instances. Although, compromising these static data structures could alter the overall system behavior, it is straightforward to protect them where their contents do not change during system runtime.
- *Non-Control Data Structures (NCDS)*; these are the data structures that maintain the lifetime of the kernel dynamic runtime objects *e.g.* processes, threads, drivers, services, tokens, modules, files. Such dynamic runtime objects change during runtime in value, location and number of running instances. It is a challenge to automatically track and protect these without relying on the OS kernel.

Locating dynamic kernel objects in memory is the most difficult step towards enabling the implementation of robust OS security solutions, as discussed above. Efficient security solutions should not rely on the OS kernel memory or APIs to extract runtime objects, as they may be compromised and give false information to the security solution. On the other hand, the complex data layout of an operating system kernel makes it difficult to uncover and check the integrity of all system kernel runtime objects, due to their volatile nature. Moreover, modifications to kernel dynamic data violate integrity constraints that in most cases cannot be extracted from OS source code. This is because the data structure syntax is controlled by the OS code while semantic meaning is controlled by runtime calling contexts. Thus, exploiting dynamic data structures will not make the OS treat the exploited structure as an invalid instance of a given type, or even detect hidden or malicious objects. For example, Windows and Linux keep track of runtime objects with the help of linked lists. A major problem with these lists is use of C null pointers [21]. Modifications to null pointers that violate intended integrity constraints cannot be extracted from source code as the violations depend on calling contexts at runtime. This makes it easy to unlink an active object by manipulating pointers and thus the object becomes invisible to the kernel and to monitoring tools that depend on kernel APIs and memory *e.g.* HookFinder [9] or memory traversal such as KOP [11], CloudSec [7], and OSck [3].

A. Related Work

Previous research efforts in the area of tracking and checking the integrity of kernel runtime objects limit themselves to the kernel static data *e.g.* system call and descriptor tables [13], or can reach only a sub-set of the dynamic kernel data [2, 14], resulting in security holes, limited protection and an inability to detect zero-day threats. The approaches of DeepScanner [10], DIMSUM [12], Gilbraltar *et al.* [2], and Petroni *et al.* [14] (all value-invariant approaches) are limited in that their authors depend on their knowledge of the kernel data layout, making their approach limited to a few structures. Also these tools do not consider the generic pointer relations between structures, making their approaches imprecise and vulnerable a wide range of attacks that can exploit the generic pointers. These approaches can have a high performance overhead due to large signatures.

To the best of our knowledge all existing approaches, whether value-invariant or memory traversal (with the exception of KOP [11], and SigGraph [6]) depend on OS expert knowledge to provide kernel data layout definitions that resolve the points-to relations between structures. SigGraph follows a systematic approach to define the kernel data layout, in order to perform brute force scanning using the value-invariant approach. However, it only resolves the direct points-to relations between data structures without the ability to solve generic pointer ambiguities, making their approach unable to generate complete and robust signatures for the kernel. KOP is the first and only tool that employs a systematic approach to solve the indirect points-to relations of the kernel data. However, KOP is limited in that: the points-to sets of the void * objects are not precise and thus they use a set of OS-specific constraints at runtime to find out the appropriate candidates for the objects. KOP assumes the ability to detect hidden objects based on traditional memory traversal techniques which are vulnerable to object hiding. Moreover, both KOP and SigGraph have a very high performance overhead when uncovering kernel runtime objects in a memory snapshot.

Rhee *et al.* [15] proposed an interesting approach to detect runtime objects by analyzing executed object allocation and reallocation instructions. Their approach has quite high performance overhead and therefore cannot be used in traditional OS security tools – only for advanced debugging tools. Also, despite the feature of detecting allocations and deallocations in near real time, they cannot identify the object type. They need to analyze executed instructions offline in order to identify object type and details.

III. DIGGER ARCHITECTURE

DIGGER's goal is to systematically uncover all kernel running objects in a memory snapshot or from a running virtual machine – without deep knowledge of the kernel data layout – in order to enable systematic integrity checks for the dynamic kernel runtime objects. The high-level process of DIGGER is shown in **Error! Reference source not found.**. DIGGER has three main components: *Static Analysis Component, Signature Extraction Component and Dynamic Memory Analysis Component,* discussed below in detail.

A. Static Analysis Component

Performing static analysis on the kernel source code is the key to automating the process of extracting kernel objects' details without any prior knowledge of the kernel data layout in memory. In this phase of the analysis, DIGGER performs static *points-to* analysis [16, 17] on the kernel's source code, allowing it to systematically solve the ambiguous *points-to* relations between kernel data structures and to infer the candidate target types and values of the generic pointers. *Points-to* analysis is the problem of determining statically a set of locations to which a given pointer may point to at runtime. *Points-to* analysis for C programs has been widely used in compiler optimization, memory error detection and program understanding [18, 19]. However, none of the previously discussed approaches meet our requirements in analyzing the kernel as they do not scale to the enormous size and complexity of a typical OS kernel. They also typically sacrifice precision for performance. In our analysis, precision is an important factor. We want the most precise points-to sets to be computed.

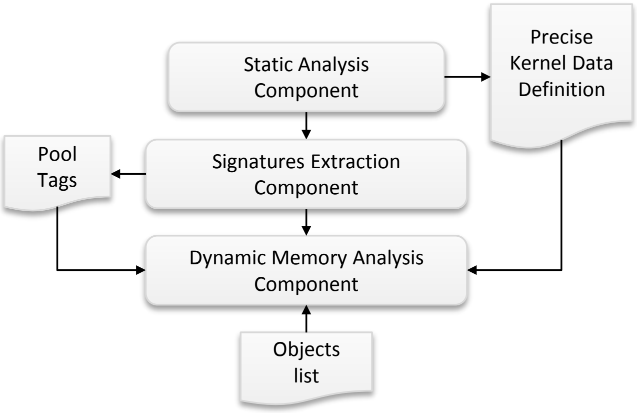

Static Analysis Component
Precise Kernel Data Definition
Pool Tags
Signatures Extraction Component
Dynamic Memory Analysis Component
Objects list

Fig. 1. The high-level process of DIGGER approach.

The result of our *points-to* analysis is a kernel data definition represented as a type-graph. This type-graph precisely models data structures and reflects accurately both direct and indirect relations that reflect the memory layout of the data structures. Fig. 2 shows a snapshot of the real computed type-graph that summarizes the different data types located in the kernel along with their connectivity patterns and reflects the inclusion-based relations between kernel data structures for both direct and indirect relations. The generated type-graph is **not** used to uncover kernel objects - it is only used to retrieve running object's details after discovering a running instance of a specific object type (details of discovering objects is discussed later in section C). The level of an object's details is selected by tool users based on the required hierarchal depth. This enables controlling the trade-off between details and performance overhead, as some object types have hundreds of hierarchically-organised fields.

We build the type-graph using our tool KDD [20, 21]. KDD is a static analysis tool that has the ability to perform inter-procedural, context-sensitive, field-sensitive and inclusion-based *points-to* analysis on the kernel source code.

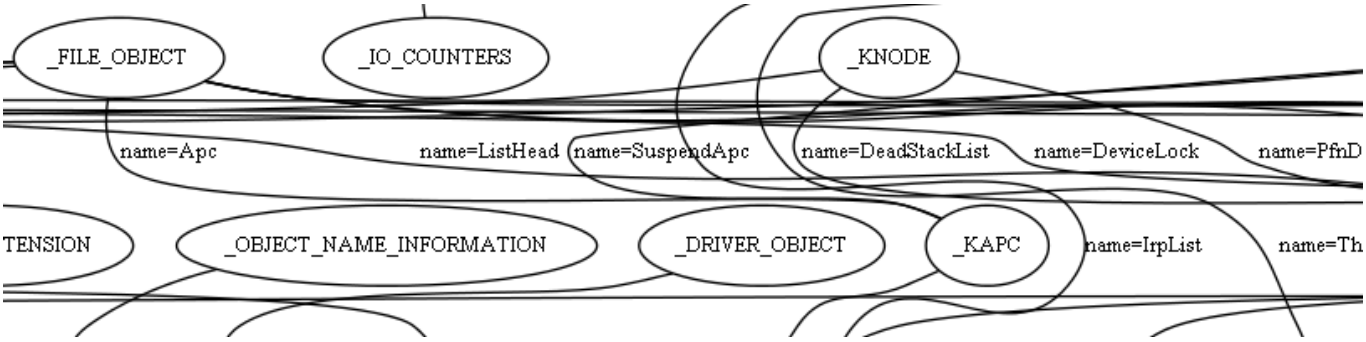


Fig. 2. A snapshot of the real computed type graph

KDD is able to perform precise and scalable *points-to* analysis for large C programs that contain millions lines of code *e.g.* operating system kernels, without any prior knowledge of the operating system structure. The type-graph is created and refined by our points-to analysis algorithm in three steps:

1) *Intraprocedural Analysis;* to perform local points-to analysis on the procedure level, but without information about caller or callees.
2) *Interprocedural Analysis;* enables performing points-to analysis across different files to perform whole-program analysis.
3) *Context-Sensitive Points-To Analysis;* enables computing complete points-to sets combined with the different call site, for the whole kernel source code.

KDD can be applied on any C-based OS *e.g.* Linux, BSD and UNIX to perform a detailed and accurate points-to analysis in order to reflect a precise memory kernel data layout offline. KDD is able to scale to the enormous size of kernel code. This scalability was achieved by using Abstract Syntax Trees (AST) as the basis for points-to analysis. The compact and syntax-free AST improves time and memory usage efficiency of the analysis. This is because instrumenting AST is more efficient than instrumenting machine code or a low-level representation such as Medium-level Intermediate Representation (MIR). Low-level representations of any source code are extremely big in size, omits very important information such as declarations, data types and type casting, and create a lot of temporary variables that are allocated identically to source code variables and thus are not easily distinguishable from source code variables [22].

B. Signature Extraction Component

The goal of this component is to extract efficient signatures for operating system data structures, to be used in discovering the runtime objects instances using the value-invariant approach. Obtaining robust signatures for kernel data structures is difficult for the following reasons:

- Data structure sizes are not small. From our analysis for Windows and Linux operating systems, we found that a single data structure could occupy several hundred bytes. Such big signatures increase discovery cost and performance overhead.
- It is difficult to identify which fields of a target data structure can be used. Dolan-Gavitt *et al.* [23] showed how to generate robust signatures for the kernel data structures by employing a feature selection process that ensures that the features chosen are those that cannot be controlled by the attacker. However, this approach is time-consuming and is not practical for systematically checking the thousands of the kernel data structures. This is because their solution profiles OS execution in order to determine the most frequently accessed fields and then tries to modify their contents to determine which are critical to the correct functioning of the system. In addition, as discussed by Liang [10], the fields relevant to the correct operations of a target kernel data structure may not be able to act as signatures because it is difficult to distinguish the target kernel structure from memory based on these fields. Second, the fields of some passive kernel objects are not relevant to system operations. The operations of such system objects still are normal even after arbitrary modification.
- The OS kernel contains thousands of data structures, making the process of generating "unique" signatures for this huge number of structures very challenging.

In order to overcome those difficulties, DIGGER makes use of the pool memory tagging schema of the kernel object manager to overcome the first two problems, and is motivated by the following paragraph from the Mark Russinovich's Windows Internals book [24] (we call it *WI-note*) to overcome the third problem – details discussed below: "*Not all data structures in the Windows operating system are objects. Only data that needs to be shared, protected, named, or made visible to user-mode programs is placed in objects. Structures used by only one component of the operating system to implement internal functions are not objects*".

1) Pool Memory Tagging Schema

Windows kernels use pool memory to allocate kernel dynamic objects. Pool memory is a form of memory manager that the kernel can use when it needs to allocate and free dynamically allocated memory objects. The pool memory can be thought of as a kernel-mode equivalent of the user-mode heap memory. When the object manager allocates a memory pool block using the allocation routine `ExAllocatePoolWithTag`, it associates the allocation with a pool tag. A pool tag[1] is a unique four-byte tag for each object type and is stored in reverse order. We use these pool tags as a value-invariant signatures to uncover the kernel objects running instances presence in memory. However, the pool tag is not enough to be an object signature. For instance, if we have a pool tag "Proc" – the pool tag for Process object type – and we scan the memory using the ASCII code of this

[1] The pool tag list for the Windows operating system can be extracted from the symbol information – Microsoft Symbols.

pool tag, any word that has the same ASCII string will be detected as a running instance from that object type – process. Thus we need to add another checking signature that guarantees accurate results and at the same time does not increase the performance overhead. We make use of the object dispatcher header to provide the additional checking signatures. Each allocated object starts with a dispatcher header that is used by the OS to provide synchronization access to resources. This structure describes mainly an object type, size and state, as shown in Fig. 3.

```
typedef struct _DISPATCHER_HEADER {
    union {
        struct {
            UCHAR Type;
            union {
                UCHAR Absolute;
                UCHAR NpxIrql;
            };
            union {
                UCHAR Size;
                UCHAR Hand;
            };
            union {
                UCHAR Inserted;
                BOOLEAN DebugActive;
            };
        };
        volatile LONG Lock;
    };

    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;
```

Fig. 3. Dispatcher header data structure in Widows OS

The first three bytes of the dispatcher header are unique for each object type, as they describe an object's type and size. These three bytes can be calculated from the generated type-graph – from our static analysis component. From our experiments, we found that those three bytes are static and cannot be changed during object runtime. Using the pool tagging schema and the first three bytes of the object header, DIGGER can discover the presence of runtime object instances in memory. Key features of using pool tags as signatures are:

- We are not constrained by data structure layout for a specific kernel build. This approach works for multiple OS kernel versions. .
- The very small size of the scanning value-invariant signature decreases performance overhead significantly.

2) *WI-Note*

To the best of our knowledge, all current OS security research (for Windows and Linux operating systems) treat all data structures (type definitions) as objects, and do not consider the WI-note. The WI-note enables filtering the list of data structures extracted at the static analysis step, in order to obtain a list of the actual runtime object types. Each data structure that has a pool tag used by the Windows memory manager is considered to be an object and the other data structures are not. This massively reduces the number of object types from thousands to dozens. This solves the problem of generating unique signatures for the huge number of kernel data structures (the third obstacle), and also frees resources for analysis of the most important data structures.

For the other data structures (non-objects that we consider less important than objects), we use a KDD-generated type-graph to traverse memory to uncover those data structures using the *points-to* relations of those data structures with the uncovered kernel objects.

C. Dynamic Memory Analysis Component

The goal of this component is to use the output of the static analysis and signature extraction components, in order to uncover the runtime objects in a memory snapshot or a running virtual machine – using virtual machine introspection techniques. The output of this component is an object-graph whose nodes are instances of data structures and objects, and edges are the relations between these objects and data structures.

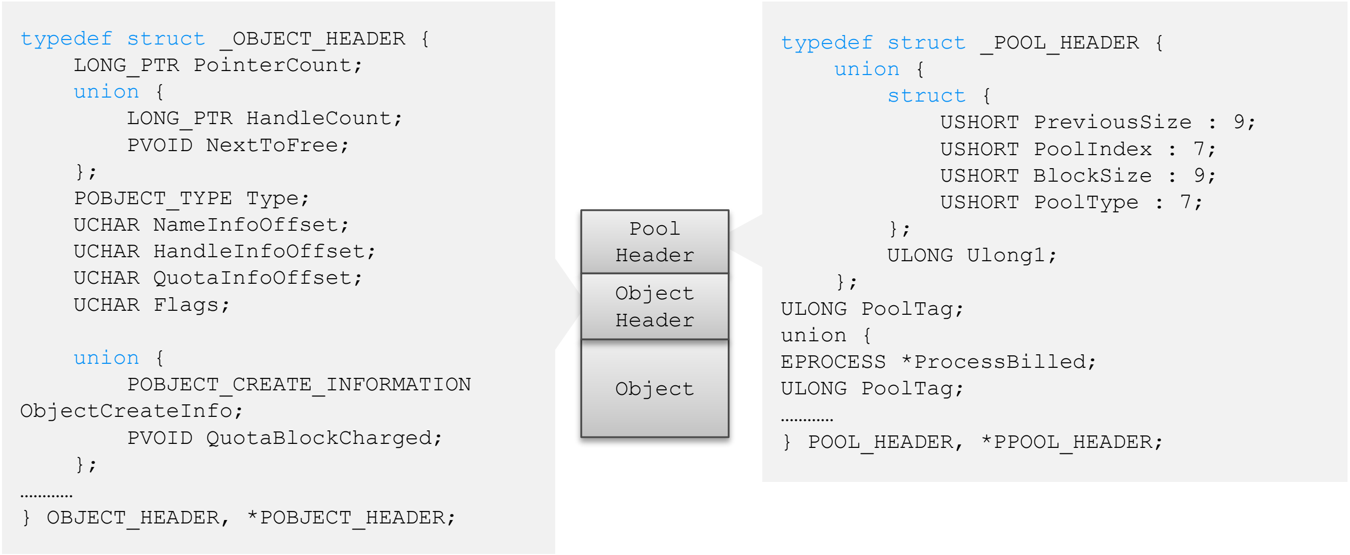
First, using the pool tags and the additional checking signature, the dynamic memory component scans the kernel address space with eight byte granularity (the smallest size of the pool memory chunk) to extract the runtime instances of the different kernel object types. Until this step is complete, we can only identify that there is a running instance of an object of type *T*, but we cannot know any details about the object itself or even the object name. So, we need to find a relation between the pool tag and the object body in order to traverse the memory to extract the object details. When an object is being allocated by the object manager it is prefixed by an object header and the whole object (including the object header) is prefixed with a pool header data structure, as shown in Fig. 4. The pool header data structure is a data structure used by the Windows object manager to keep track of memory allocations. The most important fields in the pool header – for DIGGER – are the pool tag and the block size fields. These fields help our algorithm to extract the object details as follows:

- *Pool Tag;* by subtracting the offset *f* of the pool tag field from the address *x* where an object has been detected at (using the pool tag and the additional checking signature), we can get the pool block start memory address *y*. Then, by adding the size of the pool header and the object header to address *y*, we can calculate the object's start address *O*. The size of the pool and object headers are calculated from the kernel type-graph. Having the object's start address, DIGGER can retrieve the object's details – based on KDD-generated type-graph – by traversing the kernel memory.
- *Block Size;* indicates the pool block size *s* that has been allocated for an object *O*. This field helps to speed up the scan process, by skipping *s* bytes of the kernel address space – starting from the *y* address – to reach the start address of next pool block.

The dynamic memory analysis component has two strategies for uncovering running kernel objects: *Memory Images* and *Un-Mappable Memory pages*.

1) *Complete Memory Images*

The size of a complete memory image is quite large and the kernel address space ranges from 1GB to 2GB in 32bit OSs and up to 8TB in 64bit OSs according to the memory layout used by the hardware and the available hardware memory. Scanning such a huge number of memory pages is too expensive.



```
typedef struct _OBJECT_HEADER {
    LONG_PTR PointerCount;
    union {
        LONG_PTR HandleCount;
        PVOID NextToFree;
    };
    POBJECT_TYPE Type;
    UCHAR NameInfoOffset;
    UCHAR HandleInfoOffset;
    UCHAR QuotaInfoOffset;
    UCHAR Flags;

    union {
        POBJECT_CREATE_INFORMATION
ObjectCreateInfo;
        PVOID QuotaBlockCharged;
    };
…………
} OBJECT_HEADER, *POBJECT_HEADER;
```

```
typedef struct _POOL_HEADER {
    union {
        struct {
            USHORT PreviousSize : 9;
            USHORT PoolIndex : 7;
            USHORT BlockSize : 9;
            USHORT PoolType : 7;
        };
        ULONG Ulong1;
    };
ULONG PoolTag;
union {
EPROCESS *ProcessBilled;
ULONG PoolTag;
…………
} POOL_HEADER, *PPOOL_HEADER;
```

Fig. 4. The memory layout of allocated objects in the pool memory.

```
typedef struct _POOL_DESCRIPTOR {
    POOL_TYPE PoolType;
    ULONG PoolIndex;
    ULONG RunningAllocs;
    ULONG RunningDeAllocs;
    ULONG TotalPages;
    ULONG TotalBigPages;
    ULONG Threshold;
    PVOID LockAddress;
    PVOID PendingFrees;
    LONG PendingFreeDepth;
    SIZE_T TotalBytes;
    SIZE_T Spare0;
    LIST_ENTRY ListHeads[POOL_LIST_HEADS];
} POOL_DESCRIPTOR, *PPOOL_DESCRIPTOR;
```

Fig. 5. Pool Descriptor Data Structure

To solve this problem and get the fastest coverage for the kernel address space, we scan only the pool memory instead of the whole kernel address space. At system initialization, the memory manager creates dynamically sized memory pools, and each pool is defined by a pool descriptor, shown in Fig. 5.

The Pool descriptor is a management structure that tracks pool usage and defines pool properties such as the memory type. The pool descriptor is mainly responsible for tracking the number of running allocations and deallocations since system initialization, and also helps the system to keep track of free pool chunks that can be used to allocate new objects. The most important fields – used by DIGGER's approach – are:

- *RunningAllocs;* indicates the number of allocations for all object types that have been allocated since system initialization. DIGGER makes use of this data member to validate the number of uncovered objects (all data types) at specific time *t*, in order to ensure that its results are accurate and represent the actual allocated objects.
- *RunningDeAllocs;* indicates the number of deallocations for all object types that have occurred since system initialization. DIGGER makes use of this field to validate the detected hidden objects that have not been deallocated.
- *PoolType;* defines a pool chunk type. There are two distinct types of pool memory in Windows OS: paged pool and non-paged pool. Both are used by the kernel address space to store the kernel and executive objects, respectively. The non-paged pool consists of virtual memory addresses that are guaranteed to reside in physical memory as long as the corresponding kernel objects are allocated. The kernel uses the non-paged pool memory to store the runtime objects that may be accessed when the system cannot handle page faults *e.g.* processes, threads and tokens. The paged pool consists of virtual memory that can be paged in and out of the system. This means that by scanning the non-paged pool memory, which is a trusted source of information, we can get all the running object instances that are potential targets for hackers as they always reside in physical memory. On uniprocessor or multiprocessor systems there exists only one non-paged pool. This number can be confirmed using the global variable `nt!ExpNumberOfNonPagedPools`. The OS maintains a number of global variables that define the start and end addresses of the paged and non-paged pool memory: `MmPagedPoolStart`, `MmPagedPoolEnd`, `MmNonPagedPoolStart` and `MmNonPagedPoolEnd`. These pointers can be used to speed up scanning by limiting the scanned area. From our observations, we found pool memory size takes a very small portion from the system address space, as shown in Table 1.

2) Un-mappable Memory Pages

In this case, the size of pages set is reasonably small. We perform a scan on the whole set of memory pages using the pool tag and the additional checking signature. However, as the memory mapping information may not be available in such un-mappable memory pages, not all of the discovered objects' details can be retrieved as we depend on the memory traversal technique according to the generated type-graph and this approach requires accessing multiple memory pages.

Table 1. Paged and Nonpaged Pool Memory Size.
"L" column indicates the pool memory limit that can be allocated and "A" column indicates the actual allocated memory in some operating systems we used in our experiments.

Operating System	RAM (GB)	Paged Pool (MB)		Nonpaged Pool (MB)	
		L	A	L	A
Windows XP 32-bit	2	368	17.9	262.1	4
Windows XP 64-bit	2	3498	28.1	875.5	15.3
Server 2008 32-bit	2	2000	24.5	1555	15.4
Windows 7 64-bit	8	8000	372	6238	174

IV. Implementation and Evualtion

We have developed a prototype of DIGGER. The static analysis component was built using our previously developed tool, KDD [1, 20]. We have implemented a prototype of KDD using C#. KDD uses *pycparser* [25] to generate AST files of the kernel's source code. KDD then uses the AST files to apply our points-to analysis algorithm to generate the type-graph. We have used Microsoft's Parallel Extensions to leverage multicore processors in an efficient and scalable manner to implement KDD. Threading has also been used to improve parallelization of computations (.Net supports up to 32768 threads on a 64bit platform). The signatures and runtime components are standalone programs and all components are implemented in C#. The runtime component could work: *(i)* offline on memory snapshot, raw dumps (*e.g.* dumps in the Memory Analysis Challenge and Windows crash dumps), and VMware suspended sessions. *(ii)* Online in a virtualized environment by scanning VMs' physical memory from the hypervisor level using virtual machine introspection techniques.

We have evaluated the basic functionality of DIGGER with respect to the identification of the kernel runtime objects and the performance overhead of uncovering these objects. We performed different experiments and implemented different OS security prototype tools to demonstrate DIGGER's efficiency.

A. Static Analysis Component

For the static analysis component, we applied KDD's static analysis to the source code of the Windows Research Kernel (WRK[2]) (a total of 3.5 million lines of code), and found 4747 type definitions, 1858 global variables, 1691 void pointers, 2345 null pointers, 1316 doubly linked list and 64 single linked lists. KDD took around 28 hours to complete the static analysis on a 2.5 GHz core i5 processor with 12 GB RAM. As our analysis was performed offline and just once on each kernel version, the performance overhead of analysing kernels is acceptable and does not present any problem for any security application using KDD. The performance overhead of KDD could be decreased by increasing the hardware processing capabilities, as such types of analysis usually run with at least 32 GB RAM.

B. Runtime Analysis Component

To enable efficient evaluation for the runtime component, we need a *ground truth* that specifies the exact object layout in kernel memory so that we can compare it with the results of DIGGER to measure false alarm rate. We built the ground truth as follows: we extracted all data structure instances of the running Windows OS memory image *via* program instrumentation using the Windows Debugger (WD). We instrumented the kernel to log every pool allocation and deallocation, along with the address using the WD. In particular, we modified the GFlags (Global Flags Editor) to enable advanced debugging and troubleshooting features of the pool memory. We then measured DIGGER efficiency as the fraction of the total allocated objects that DIGGER was able to identify correctly (i.e. the correct object type).

We performed experiments on 3 different versions of the Windows OS on a 2.8 GHz CPU with 2GB RAM. Table 2 shows the results of DIGGER and the Windows Debugger in discovering the allocated instances for some object types in two of the three Windows versions. From Table 2 we can see that DIGGER achieves zero false negative rates (FN), and a low false positive rate (FP). However, from our manual analysis of the results, we found that this reported false positive rate is not an actual false positive. This difference represents deallocated objects that still persist in the physical memory after termination; we call these "dead memory pages objects – DMAO". These objects are present because the Windows operating system does not immediately clear the contents of deallocated memory pages (thereby delaying the overhead of writing zeroes to physical memory). We propose that whenever the kernel has to allocate a new object it will return the pool block address from the pool free list head. For example, the EPROCESS structure of a newly created process will overwrite the object data of a process that has been terminated previously. This makes sense because when a block is freed using the `free` function call, the allocator just adds the block to the list of free blocks without overwriting memory.

We noticed (from our analysis) that the pointer and handle count of the DMAO is always zero. This enables us to differentiae between the active objects from the DMAO, and thus our actual false positive rate (FP*) becomes zero. Those DMAOs can provide forensic information about an attacker's activity. Imagine that an attacker runs stealthy malware and then terminates it on a victim's machine. After termination there may still exist for a non-trivial period of time some forensic data of interest in the dead memory pages. To prove our assumption, we analyzed the dead memory pages in order to uncover semantic data of interest for the some terminated processes. However, our approach could work for any other object type. We used some benchmark programs to run in three memory images and then analyzed the dead memory pages to uncover some data of interest: user login information (GroupWise email client), chat sessions (Yahoo messenger), FTP sessions (FileZilla).

We created 9 processes (three of these were benchmark programs) and performed some CPU-intensive operations using these processes. We terminated these processes after 5 hours, 2 hour and 15 minutes in three different memory images – identified L, M and S, respectively. Then we created 4 different (new) processes 5 minutes after termination. The memory images were then scanned for runtime objects using DIGGER's runtime component.

[2] WRK is the only available source code for Windows.

Table 2. Experimental results of DIGGER and WD on Windows XP 32 bit and 64bit.
Memory, paged and nonpaged columns reprsent the size in pages (0x1000 graunrality) of the kernel address space, paged pool and nonpaged pool, repectively. WD and DIG refer to WD's and DIGGER results. FN, FP and FP* denote the false negative, reported false positive and the actual false poitive rates, repectively.

Object	Windows XP 32bit					Windows XP 64bit				
	Memory		Paged		Nonpaged	Memory		Paged		Nonpaged
	915255		27493		11741	1830000		35093		17231
	WD	DIGGER	FN (%)	FP (%)	FP* (%)	WD	DIGGER	FN (%)	FP (%)	FP* (%)
Process	119	121	0.00	1.65	0.00	125	125	0.00	0.00	0.00
Thread	2032	2041	0.00	0.44	0.00	2120	2121	0.00	0.04	0.00
Driver	243	243	0.00	0.0	0.00	211	211	0.00	0.00	0.00
Mutant	1582	1582	0.00	0.0	0.00	1609	1609	0.00	0.00	0.00
Port	500	501	0.00	0.19	0.00	542	542	0.00	0.00	0.00

We found that three processes from the terminated processes' physical addresses were overwritten by the EPROCESS structure for new processes, while another three processes (from the terminated ones) still persisted in memory (at the same address in the memory). Thus we make the following observations. First, for the email client we were not able to identify the login information (user name and password) for all of the memory images. For the ftp client we were able to identify the server name, and the server and client connection ports for the S image only, without any ability to locate the login credentials in all of the three images. For the chat benchmark application, we were able to locate the username, the connection port and few chat sessions in the S image only. This data recovery approach is not effective if the program zeros its memory pages before termination.

C. *Performance Overhead*

We have evaluated DIGGER's runtime performance to demonstrate that it can perform its memory analysis in a reasonable amount of time. We measured DIGGER's running time when analyzing the memory snapshots used in our experiments. The median running time was around 0.8 minutes to uncover 12 different object types from the nonpaged pool, and 1.6 minutes to uncover another 15 object type from the paged pool. This time included the time of loading the memory snapshot from the disk to the runtime analysis component. We consider this running time to be acceptable for offline analysis and even for online analysis in virtualized environments. This is because DIGGER is able to detect the DMAOs that could be created and terminated between the scan time intervals. However, we cannot argue that it would be 100% accurate. Comparing DIGGER with SigGraph [6], DIMSUM [12], KOP [11], CloudSec [7]: DIGGER is the fastest with highest coverage and lowest performance overhead. The performance overhead of extracting object details based on our generated type-graph differs according the required details-depth. Fig. 6 shows the time consumed (in seconds) to extract object details with different depths for all of the running instances from specific object types. "I" denotes the number of the running objects from the object, and "D" denotes the depth of the extracted details.

V. Security Applications

We have further evaluated our DIGGER approach by developing three prototype OS security tools to demonstrate its efficiency and applicability. These are (i) a generic hidden objects detection tool, (ii) a brute force scanning tool, and (iii) an external VM monitoring tool. We chose these applications because they address common important OS security activities. Our experiments with these tools have demonstrated DIGGER's efficiency and the false alarm rate is similar to that shown in table 2.

A. *Detecting Object Hiding Attacks*

Previous efforts – in the area of detecting object hiding attacks – have focused on detecting specific types of hidden object types by hard-coding OS expert knowledge of the memory kernel data layout (e.g. Petroni *et al.* [14] and Baliga *et al.* [2]). Other approaches (e.g. Nanavati *et al.* [26]) rely on value-invariants such as the matching of the process list with the thread scheduler, and limiting their approach to detect only hidden processes. Other approaches (e.g. Riley *et al.* [27] and Xuan *et al.* [28]) are based on logging malware memory accesses and provide temporal information about operating system behavior. However these can only cover known attacks and cannot properly handle zero-day threats. There are some approaches (e.g. Antfarm [29] and Wen *et al.* [30]) that track the value of the CR3 register that holds the process table address. Although this approach is useful in a live environment for detecting hidden processes, it cannot be used for memory forensics applications. It also has a very high performance overhead. In summary, all of the previous approaches are time-consuming, and require a human expert with deep knowledge of the operating system to create the rules and thus cannot cover all system objects and enable systematic discover of all hidden object types.

Given DIGGER's ability to uncover kernel objects, we developed a tool called D-Hide that can systematically uncover all kinds of stealthy malware (not just limited to specific object type, as done to date), by detecting their presence in physical memory. We used DIGGER's approach to uncover runtime kernel objects, and then performed an "external" cross-view comparison with the information retrieved from mapping the physical memory using our generated type-graph.

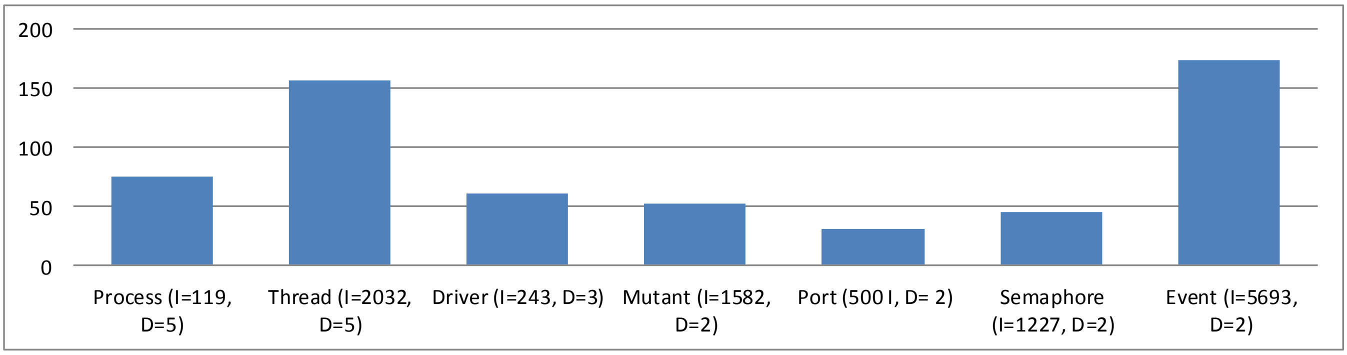


Fig. 6. Object details extraction normalized time.

In other words, in order to enable cross-view detection we needed two different views to compare. In D-Hide the first view was DIGGER's view and the other view was a traditional memory traversal view; starting from the operating system global variables and then follow pointer dereferencing until it covered all kernel running objects. Any discrepancy in this comparison revealed hidden kernel objects. We implemented a traditional memory traversal add-on for the runtime component that takes our generated type-graph and based on that graph, it traverses the kernel address space. We evaluated D-Hide's ability to identify hidden objects with four real-world kernel rootkit samples: *FURootkit*, *FuToRootkit*, *AFX Rootkit* and HideToolz. We also used *WinObj* (a windows internal tool) to compare the results with D-hide. D-hide correctly identified all hidden objects with zero false alarms. D-Hide has three key advantages:

- No need for deep knowledge of the runtime kernel data layout, as it depends on DIGGER's static component to get an accurate definition of kernel data layout.
- D-Hide can perform cross-view comparisons without the need for any internal tools *e.g.* task manager or WinObj that get the internal view, as done in the current cross-view research [31]. This feature enables deploying D-Hide in VMs hosted in the cloud platform where the cloud providers do not have any control over VMs, as discussed in [1, 13].
- D-Hide is unlike previous tools [26, 32] that rely on the authors' knowledge of the kernel data and thus is not limited to specific objects.

B. *A Brute Force Scanning Tool*

Given a range of memory addresses and a signature for a data structure or object, brute force scanning tools can detect if an instance of the corresponding data structure exists in the memory range or not [6]. Brute force scanning of kernel memory images is an important function in many operating system security and forensics applications, used to uncover semantic information of interest *e.g.* passwords, hidden processes and browsing history from raw memory.

Given DIGGER's ability to uncover kernel objects, we developed B-Force – a brute force scanning tool. We mainly depended on the pool memory tagging schema to detect instances of corresponding data structures existing in a set of memory pages. From our experiments with five different small crash dumps of small sizes ranging from 12MB to 800MB, we found that this method is effective and has no false alarms. This method could reveal false positives if the memory page set does not contain the pool header (that contains the pool tag) of the pool block along with the first three bytes of the object itself (to perform the additional signature checking). However, from our point of view this is unlikely, as the single memory page size is big enough to contain tens of pool blocks.

C. *Virtual Machine Monitoring*

To the best of our knowledge, all current Virtual Machine Introspection (VMI) research [8, 33-36] depend on manual efforts to build a kernel data definition to solve the semantic gap. XenAccess [35] depends on manual efforts to build a data definition to overcome the semantic gap for specific data structures. PsycoTrace [37] follows a similar approach, as does KvmSec [38] and VIX Tools [8], X-Spy [31], VMwatcher [39] and SIM [40]. Security research targeting VMs hosted on the IaaS platform is relatively limited. Most of current approaches [41, 42] depend on deploying traditional in-guest security solutions inside the VMs. However, some researchers [13, 43] have discussed the complexities of the IaaS platform and the challenges of implementing security solutions for it.

We modified our earlier-developed VM monitoring tool, *CloudSec* [7], to use DIGGER's approach instead of the manually built kernel data definitions. *CloudSec* monitors a VM's memory from outside the VM itself, without installing any security code inside the VM, to provide fine-grained inspection of the VM's physical memory. *CloudSec* actively reconstructs externally a high-level semantic view of the running OS kernel data structure instances for the monitored VMs' OS in order to overcome the semantic gap problem.

Using our new DIGGER approach, we extended *CloudSec* to map the physical memory of a VM running Windows XP 64bit. To evaluate the mapping results, we compared the results with the internal OS view using the Windows Debugger. *CloudSec* successfully uncovered and correctly identified the running kernel objects, with zero false alarms. The performance overhead of *CloudSec* to uncover the entire kernel running objects was around 1.1, 1.9 and 2.8 minutes with 0-level, 1-level and 2-level depths, respectively for a VM with a 2.8 GHz CPU and 4GB RAM. The VM was executed under a normal workload (50 processes, 912 threads). We can see that the performance overhead of scanning a VM's memory online is less that scanning a memory image, as access to VM's memory *via* hypervisors is faster than uploading a memory image to the analysis tool.

VI. Discussion

DIGGER's approach provides a robust view of OS kernel objects not affected by the manipulation of actual kernel memory content. This enables development of different OS security applications as discussed in section V, in addition to enabling systematic kernel data integrity checks based on the resultant object-graph. A key feature of DIGGER is its systematic approach it extracting OS kernel data layout and to disambiguate the points-to relations between data structures, without any prior knowledge of the OS kernel memory data layout. Performing static analysis on kernel source code to extract robust type definitions for the kernel data structures has several advantages. It minimizes the performance overhead in security applications as a major part of the analysis process is done offline. If no static analysis were done, every pointer dereference would have to be instrumented, which increases performance overhead. It also maximizes the likelihood of detecting zero-day threats that target generic pointers. This is because kernel objects and associated data structures recovered by DIGGER can be checked against their statically analysed possible types and structures. Even if an attack has never been seen before, DIGGER allows us to determine if an invalid kernel object structure is present in the running OS kernel. The robust and quite small signature size used by DIGGER to uncover runtime objects enhances performance. It allows a security solution employing our DIGGER approach to determine a kernel object type and structure fast enough for real-time usage e.g. as in *CloudSec*, as well as brute-force scanning, as in *B-Force*.

As the pool memory concept is only related to Windows operating systems, the current approach used in DIGGER's runtime component can only be used to analyze Windows operating systems, including all its kernel builds except Windows vista. DIGGER's runtime component is not related to a specific version of the Windows OS kernel and can work on either 32-bit or 64-bit layouts. However, the same approach could be used in Linux using the *slab allocation* concept. Slab allocation can be thought of as a pool memory equivalent of the Windows OS. Slab allocation is a memory management mechanism for Linux and UNIX OSes for allocating kernel runtime objects efficiently. The basic idea behind the slab allocator is having caches (similar to the pool blocks in Windows OS) of commonly used objects kept in an initialized state. The slab allocator caches the freed object so that the basic structure is preserved between uses to be used by a newly allocated object of the same type. The slab allocator consists of caches that are linked together on a doubly linked list called a cache chain that is similar to the list head of the pool memory used in Windows kernel.

Key future work extensions we are investigating include the application of the DIGGER to LINUX kernels using the slab allocation concept. Another key direction for future research is the run-time analysis of kernel data structure integrity, that we wish to extend from our *CloudSec* and *B-Force* prototypes. Another area for investigation is using function pointer disambiguation in conjunction with kernel object discovery to check for tampering at run-time.

VII. Summary

Current state-of-the-art tools are limited in their ability to accurately uncover the running instances of kernel dynamic objects. This results in limited protection and an inability to detect zero-day threats. In this paper, we presented DIGGER, a new approach that enables uncovering dynamic kernel objects with nearly complete coverage and accurate results by leveraging a set of new techniques in both static and runtime components. Our evaluation of DIGGER has shown its effectiveness in uncovering system objects and in supporting the development of several OS security solutions.

Acknowledgment

The authors are grateful to Swinburne University of Technology and the FRST Software Process and Product Improvement project for support for this research. Scholarship support for the first and fourth authors from Swinburne University of Technology is gratefully acknowledged.

References

[1] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "Supporting Virtualization-Aware Security Solutions using a Systematic Approach to Overcome the Semantic Gap," in *Proc. of 5th IEEE International Conference on Cloud Computing*, Hawaii, USA, 2012.

[2] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *Proc of 2008 Annual Computer Security Applications Conference*, 2008, pp. 77-86.

[3] O. S. Hofmann, A. M. Dunn, and S. Kim, "Ensuring operating system kernel integrity with OSck," in *Proc. of 16th international conference on Architectural support for programming languages and operating systems*, California, USA, 2011, pp. 279-290.

[4] S. Andreas, "Searching for processes and threads in Microsoft Windows memory dumps," *Digital Investigation,* vol. 3, pp. 10-16, 2006.

[5] J. Solomon, E. Huebner, D. Bem, and M. Szeżynska, "User data persistence in physical memory," *Digital Investigation,* vol. 4, pp. 68-72, 2007.

[6] Z. Lin, J. Rhee, and X. Zhang, "SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures," in *Proc. of 18th Network and Distributed System Security Symposium*, San Diego, CA, 2011.

[7] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "CloudSec: A Security Monitoring Appliance for Virtual Machines in the IaaS Cloud Model," in *Proc. of 2011 International Conference on Network and System Security (NSS 2011)*, Milan, Italy, 2011.

[8] K. Nance, M. Bishop, and B. Hay, "Virtual Machine Introspection: Observation or Interference?," *Journal of IEEE Security and Privacy,* vol. 6, pp. 32-37, 2008.

[9] H. Yin, Z. Liang, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," in *Network and Distributed Systems Security Symposium (NDSS)*, 2008.

[10] B. Liang, W. You, W. Shi, and Z. Liang, "Detecting stealthy malware with inter-structure and imported signatures," in *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security*, Hong Kong, China, 2011, pp. 217-227.

[11] M. Carbone, W. Cui, L. Lu, and W. Lee, "Mapping kernel objects to enable systematic integrity checking," in *Proc of 16th ACM conference on Computer and communications security*, Chicago, USA, 2009, pp. 555-565.

[12] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "Discovering Semantic Data of Interest from Un-mappable Memory with Confidence," in *Proc. of the 19th Network and Distributed System Security Symposium (NDSS'12)*, San Diego, CA, 2012

[13] A. S. Ibrahim, J. Hamlyn-Harris, and J. Grundy, "Emerging Security Challenges of Cloud Virtual Infrastructure," in *Proc. of 2010 Asia Pacific Cloud Workshop co-located with APSEC2010*, Sydney, Australia, 2010.

[14] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proc. of 15th conference on USENIX Security Symposium - Volume 15*, Vancouver, Canada, 2006.

[15] J. Rhee, R. Riley, and D. Xu, "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory," in *Proc. of 13th international conference on Recent advances in intrusion detection*, Ontario, Canada, 2010, pp. 178-197.

[16] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proc. of 2007 ACM SIGPLAN conference on Programming language design and implementation*, California, USA, 2007, pp. 278-289.

[17] G. Xu, A. Rountev, and M. Sridharan, "Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis," presented at the Proceedings of the 23rd European Conference on ECOOP 2009 --- Object-Oriented Programming, Italy, 2009.

[18] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proc. of ACM SIGPLAN 2004 conference on Programming language design and implementation*, Washington DC, USA, 2004, pp. 131-144.

[19] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using CLA: a million lines of C code in a second," in *Proc. of ACM SIGPLAN 2001 conference on Programming language design and implementation*, Utah, USA, 2001, pp. 254-263.

[20] A. S. Ibrahim, J. C. Grundy, J. Hamlyn-Harris, and M. Almorsy, "Supporting Operating System Kernel Data Disambiguation using Points-to Analysis," in *Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, 2012.

[21] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "Operating System Kernel Data Disambiguation to Support Security Analysis"," in *Proc. of 6th International Conference on Network and System Security (NSS 2012)*, Fujian, China, 2012.

[22] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive," in *Proc. of 5th International Workshop on Information Hiding* 2003, pp. 400-414.

[23] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proc. of 16th ACM conference on Computer and communications security*, Illinois, USA, 2009, pp. 566-577.

[24] M. Russinovich, D. Solomon, and A. Ionescu, *Windows Internals, 5th Edition*: Microsoft Press, 2009.

[25] E. Bendersky, "pycparser: C parser and AST generator written in Python " 2011, Available at http://code.google.com/p/pycparser/.

[26] M. Nanavati and B. Kothari, "Hidden Processes Detection using the PspCidTable," MIEL Labs2010, Accessed November 2010.

[27] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proc. of the 4th ACM European conference on Computer systems*, Nuremberg, Germany, 2009, pp. 47-60.

[28] C. Xuan, J. Copeland, and R. Beyah, "Toward Revealing Kernel Malware Behavior in Virtual Execution Environments," in *Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection*, Saint-Malo, France, 2009, pp. 304-325.

[29] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Antfarm: tracking processes in a virtual machine environment," in *Proc. of the annual conference on USENIX '06 Annual Technical Conference*, Boston, MA, 2006, pp. 1-1.

[30] W. Yan, Z. Jinjing, and W. Huaimin, "Implicit Detection of Hidden Processes with a Local-Booted Virtual Machine," in *Information Security and Assurance, 2008. ISA 2008. International Conference on*, 2008, pp. 150-155.

[31] B. Jansen, H. Ramasamy, and M. Schunter, "Architecting Dependable and Secure Systems Using Virtualization," *Architecting Dependable Systems,* pp. 124-149, 2008.

[32] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Seattle, WA, USA, 2008, pp. 91-100.

[33] J. Pfoh, C. Schneider, and C. Eckert, "Exploiting the x86 Architecture to Derive Virtual Machine State Information," in *Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on*, 2010, pp. 166-175.

[34] T. Garfinkel and M. Rosenblum, "Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. of 2003 Network and Distributed Systems Security Symposium*, 2003, pp. 191-206.

[35] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, 2008, pp. 233-247.

[36] Adit Ranadive, Ada Gavrilovska and Karsten Schwan, "IBMon: monitoring VMM-bypass capable InfiniBand devices using memory introspection," in *3rd ACM Workshop on System-level Virtualization for High Performance Computing*, Nuremburg, Germany, 2009, pp. 25-32.

[37] F. Baiardi, D. Maggiari, and D. Sgandurra, "PsycoTrace: Virtual and Transparent Monitoring of a Process Self," in *Proc. of17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Weimar, 2009, pp. 393-397.

[38] F. Lombardi and R. D. Pietro, "KvmSec: a security extension for Linux kernel virtual machines," in *Proc. of 2009 ACM symposium on Applied Computing*, Honolulu, Hawaii, 2009, pp. 2029-2034.

[39] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proc. of 14th ACM conference on Computer and communications security*, Virginia, USA, 2007, pp. 128-138.

[40] Monirul I. Sharif, Wenke Lee, Weidong Cui, et al., "Secure in-VM monitoring using hardware virtualization," in *Proc of The 16th ACM conference on Computer and communications security*, Chicago, Illinois, USA, 2009, pp. 477-487.

[41] A. Dastjerdi and K. A. Bakar, "Distributed Intrusion Detection in Clouds Using Mobile Agents," in *Proc. of Third International Conference on Advanced Engineering Computing and Applications in Sciences*, 2009, pp. 175-180.

[42] J. Tiejun and W. Xiaogang, "The Construction and Realization of the Intelligent NIPS Based on the Cloud Security," in *Proc. of 1st International Conference on Information Science and Engineering*, Nanjing 2009, pp. 1885 - 1888.

[43] M. Christodorescu, R. Sailer, and D. L. Schales, "Cloud security is not (just) virtualization security," in *Proc. of 2009 ACM workshop on Cloud computing security*, Illinois, USA, 2009, pp. 97-102.

Amani S. Ibrahim is a PhD student at Swinburne University of Technology. Amani received the MSc degree in computer science from Ain Shams University in 2009.

She is interested in cloud computing, virtualization security, memory and run-time malwares, and operating system kernel security.

John Grundy is Professor of Software Engineering and Head of Computer Science and Software Engineering at the Swinburne University of Technology.

He has published over 230 refereed papers in areas including Automated Software Engineering, Cloud Computing, Model-driven Development, Software Methods and Tools, Software Architectures and Visual Languages.

James Hamlyn-Harris completed a Bachelor of Applied Science in 1984, a Master of Applied Science in 1986, a PhD in Engineering in 1992, a Master of Information Technology in 2006 and a Graduate Certificate of eForensics in 2011.

He is a Lecturer in computer security, computer forensics and computer programming in Swinburne University's Centre for Computing and Engineering Software Systems (SUCCESS) at Swinburne University of Technology at Hawthorn in Victoria, Australia.

Mohamed Almorsy is a PhD student at Swinburne University of Technology. Mohamed received his MSc degree in computer science from Ain Shams University in 2009.

Mohamed worked in software industry for more than seven years with three year project management experience.

He is interested in cloud computing, Adaptive security, software engineering, and project management.