In proceedings of 5th IEEE Conference on Cloud computing (CLoud 2012), 24-29 June, Waikiki, Hawaii, USA, © IEEE 2012

Supporting Virtualization-Aware Security Solutions using a Systematic Approach to Overcome the Semantic Gap

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—**A prerequisite to implementing virtualization-aware security solutions is to solve the "semantic gap" problem. Current approaches require a deep knowledge of the kernel data to manually solve the semantic gap. However, kernel data is very complex; an Operating System (OS) kernel contains thousands of data structures that have direct and indirect (pointer) relations between each other with no explicit integrity constraints. This complexity makes it impractical to use manual methods. In this paper, we present a new solution to systematically and efficiently solve the semantic gap for any OS, without any prior knowledge of the OS. We present: *(i)* KDD, a tool that systematically builds a precise kernel data definition for any C-based OS such as Windows and Linux. KDD generates this definition by performing points-to analysis on the kernel's source code to disambiguate the pointer relations. *(ii)* SVA, a security appliance that solves the semantic gap based on the generated definition, to systematically and externally map the virtual machines' physical memory and extract the runtime dynamic objects. We have implemented prototypes for KDD and SVA, and have performed different experiments to prove their effectiveness.**

Keywords - Kernel data structures; semantic gap; points-to analysis; IaaS; virtualization-aware security solutions.

I. INTRODUCTION

Infrastructure-as-a-Service (IaaS) is characterized by the concept of resource virtualization that enables running multiple Virtual Machines (VMs) on the same physical server. These VMs are hosted by the Cloud Provider (CP), but controlled by the Cloud Consumer (CC), making security a shared responsibility between CC and CP. This makes VMs a real source of security threats on the virtual infrastructure of the IaaS platform [1]. Recently, Common Vulnerabilities and Exposures (CVE) has reported multiple resource sharing exploits in the Xen and ESX hypervisors [2, 3], caused by hosted VMs. Thus, the hosted VMs cannot be trusted from the CPs' perspective to host their supported security software, as VMs can be compromised easily.

Although in-guest security solutions have the ability to get high-level information about the Operating System (OS), they are unreliable, opaque to the user and can be subverted by advanced malware, even if the security software is installed in ring 0. This raises the need for new Virtualization-Aware Security Solutions (V-ASSs) that can provide security for VMs, without installing any security software inside the VM. The virtualization supported by IaaS helps utilizing the Virtual Machine Introspection (VMI) techniques [4] that enable monitoring the hosted VMs externally at the hypervisor level. However, only hardware bytes (*e.g.* physical memory pages) can be observed in this way. This is in contrast to the internal view of the VM, where we can view high-level entities such as processes, I/O requests, and system calls, causing a "semantic gap" problem. Current research [5-7] has depended on researchers' knowledge of the OS's kernel data to manually solve the semantic gap, as solving the semantic gap requires a deep understanding of the kernel data to accurately map between the underlying hardware memory layout and kernel data structures layout. Kernel data structures are very complex; an OS kernel contains thousands of data structures with direct and indirect (pointer-based) relations between each other, with no explicit integrity constraints. In Linux and Windows, based on our observations, we found that nearly 40% of the inter-data structure relations are pointer-based (indirect) relations, and 35% of these pointer-based relations are generic pointers (*e.g.* null pointers that do not have values, and void pointers that do not have associated type declarations in the source code). Generic pointers get their values/types only at runtime according to the different calling contexts. These complexities result in an inability to cover all kernel data structures and thus reduce the efficiency of the V-ASSs, making the manual approach inadequate.

In this paper, we address the problem of how to systematically and accurately solve the semantic gap for any OS, whatever the memory layout of the hardware, and without any prior knowledge with the OS. We present: *(i) KDD (Kernel Data Disambiguator);* a tool that systematically generates a precise kernel data definition for any C-based OS (*e.g.* Windows and Linux), to enable accurate mapping of a VM's physical memory. KDD takes the source code of OS's kernel as input and outputs an accurate kernel data definition that reflects direct relations between structures and resolves the ambiguities of the pointer-based relations. KDD performs static *points-to analysis* on kernel's source code, to infer the appropriate candidate types/values for generic pointers. We designed and implemented a new points-to analysis algorithm that has the ability to provide interprocedural context-sensitive and field-sensitive points-to analysis for large programs that

contains millions lines of code *e.g.* kernels. We have implemented a prototype system for KDD and evaluated it on Linux kernel v3.0.22 and Windows Research Kernel[1] (WRK), in order to prove its effectiveness. *(ii) SVA (Security Virtual Appliance);* a V-ASS that systematically solves the semantic gap based on the generated kernel data definition, to map the physical memory of VMs efficiently. SVA utilizes VMI technique to provide fine-grained inspection of the VM's physical memory, at hypervisor level without installing any supporting code inside the VM. SVA actively reconstructs the dynamically changing kernel data structure instances (kernel dynamic objects), in order to enable effective and systematic protection for kernel data structures. We have implemented a proof-of-concept prototype for SVA and evaluated it to prove its efficiency.

In section II, we give a background of VMI, pointer problems in OSs, and we review key related work. Section III and IV discuss in details KDD and SVA, respectively. In section V we explore the implementation and evaluation details of KDD and SVA. Section VI discusses the pros and cons of our system. Finally, we summarize our conclusions.

II. BACKGROUND

VMI enables isolating the security solution from the other server workload by deploying it in a dedicated VM. This makes it difficult for hackers to detect the installed security software. Moreover, external monitoring gives the security software complete control over the hosted VMs including OS, hardware, and running software. To make VMI useful for security monitoring, it is necessary to translate the hardware bytes to actual running OS information. Such translation requires accurate mapping between kernel data structures layout, and the hardware memory layout. Such mapping is not a trivial-task for C-based OSs *e.g.* Windows and Linux. These use structures heavily to model objects and manage memory, and also use pointers extensively to simulate call-by-reference semantics, avoid expensive copying of large objects, implement lists, trees and other complex data structures, and as references to objects allocated dynamically on the heap [8]. This makes the analysis of kernel data challenging; further complicated by the fact that kernel data is implementation-dependant. Therefore, imprecise analysis will result in improper assumptions about kernel indirect relations.

A. Generic Pointers

For a better illustration to pointers problem in C-based OSs, we will use the code snippet in figure 1. We discuss in this example the context of two problems we need to address: *(i) void pointers;* the problem with void * is that the target object can only be identified during system runtime. From our example, *UniqueProcessId* is void *, however if we analyze the code, we find that it indirectly points to another data structure, *_ExHandle via* the function *ExHandler()*. We need to identify offline the set of locations that such void * could point to during runtime to enable accurate mapping for the VM's memory. *(ii) Null Pointers;* these are used for example to implement doubly-linked lists (DLs) which are heavily used in OS kernels. A DL is data structure that contains two null * fields of type DL that are used to point to the previous and next objects structured at the same list. The C definition makes a DL points to itself, but actually during system runtime it points to a specific object type according to the calling context. Procedure *Updatelinks*, from our example, is widely used in OSs to update a DL that contains dynamic objects. The problem is that the objects structured in a DL can be recognized only during runtime. Identifying the object type that a DL may hold during offline analysis helps significantly in mapping physical memory correctly.

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
   struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER;
typedef struct _EPROCESS {
   void* UniqueProcessId;
   LIST_ENTRY ActiveProcessLinks;
} EPROCESS, *PEPROCESS;
typedef struct _ExHandle {
   int* handle;
} ExHandl;
LIST_ENTRY PsActiveProcessHead;
PEPROCESS ActiveProcess;
PEPROCESS AllocatePrMemory(){
   return (PEPROCESS) malloc(sizeof(EPROCESS));
}
void CreateProcess(PEPROCESS p_ptr) {
   p_ptr = (PEPROCESS)AllocatePrMemory();
   ActiveProcess = p_ptr;
   p_ptr->UniqueProcessId=ExHandler(ActiveProcess);
   updatelinks(&ptr->ActiveProcessLinks,
   &PsActiveProcessHead);
   …
}
void* ExHandler() {
   _ExHandle tempHandle;
   tempHandle.handle = CreateHandler();
   …
   return tempHandle.handle;
}
   void updatelinks(PLIST_ENTRY src, PLIST_ENTRY tgt) {
   src->Flink = tgt->Flink; tgt->Blink = src->Blink;
}
…
```

Figure 1. Example in C showing the generic pointers problem.

B. Points-to Analysis

The goal of points-to analysis is to statically compute a set of locations to which a pointer may point to during runtime. Points-to analysis of C programs mainly differ in how we group alias information. There are two main algorithms to group alias information: Andersen's [9] and Steensgaard's [10]. Figure 2 shows a C code fragment and the points-to sets computed by those algorithms. Anderson's approach creates a node for each variable and the node may

[1] Windows is commodity OS; WRK is the only available source code for it. WRK packages core XP x64/Server 2003 kernel. This NT kernel is nearly the same in all Windows versions from Windows 2000 to 7 except Vista.

have different edges, Steensgaard's groups alias sets in one node and each node just have one edge. Andersen's is the slowest but the most precise and Steensgaard's is the fast but imprecise. Based on these approaches there are different types of analysis aspects that make the tradeoff between performance and precision: *(i) Field-Sensitivity*; distinguishing the different fields inside objects. *(ii) Context-Sensitivity*; distinguishing heap objects created through different call sites. Context-sensitive algorithms are precise, but slow in performance and complicated to be implemented. *(iii) Flow-Sensitivity*; considering the effects of pointer assignments with respect to the call-graph.

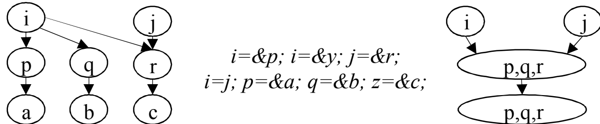

Figure 2. Alias information grouping by Steensgaard and Andersen.

Points-to analysis has been widely used in memory error detection, program understanding and complier optimization [11-13]. However, none of these approaches meet our requirements in analysing the kernel, as these approaches do not scale to the enormous size and complexity of OS's kernel. They also sacrifice precision for performance. In KDD, precision is an important factor; we want the most precise points-to sets to be computed. As the analysis is done offline and just once for each kernel version, performance is not such an important factor. KDD performs the analysis based on the Abstract Syntax Tree (AST) as a high-level representation for the kernel source code. AST captures essential structure that reflects the semantic structure of a program code while omitting unnecessary syntactic details.

C. *Related Work*

To the best of our knowledge, all current VMI research has depended on manual efforts to build a kernel data definition to solve the semantic gap. XenAccess [7] depends on the manual efforts to build a data definition to overcome the semantic gap for specific data structures. PsycoTrace [14] follows a similar approach, and the same for KvmSec [15] and VIX Tools [6]. X-Spy [16], VMwatcher [17] and SIM [18] install security code inside the VMs to get the internal view. Security research targeting VMs hosted in the IaaS platform is relatively limited. Most of current approaches [19, 20] depend on deploying traditional in-guest security solutions inside the VMs. However, some researchers [1, 21] have discussed the complexities of the IaaS platform and the challenges of implementing security solutions for it. Virtual Appliance technology has had little attention to date in academic research. However, it is used widely by security vendors *e.g.* McAfee to deploy the security solutions for IaaS platforms.

Pointer analysis algorithms for C programs have been studied intensively over the last two decades [11-13]. Their use has predominantly been for compiler optimizations and their main goal has thus been performance. Some work has attempted performing field and context sensitivity analysis on large programs [22, 23]. However none has been shown to scale to large programs *e.g.* OS's kernel code with a high precision rate. Yu *et al.* [23] proposed a context and field sensitive pointer analysis based on the static single assignment (SSA) form that gets the points-to information for variables only, but not for structures. Hardekopf *et al.* [24] proposed a flow-sensitive pointer analysis approach but they did not consider the generic pointers problems of the indirect dereferencing. Heintze [12] proposed a field-sensitive, context-insensitive pointer analysis algorithm that is based on dynamic transitive closure. This assumes that an edge between two variables must be a non-null path (which does not solve the generic pointers problem). Several pointer analysis algorithms are context-sensitive [13, 25]. However, these algorithms are used during program compilation to name objects by allocation site, not by the access path, which do not solve the null pointers ambiguity.

III. SEMANTIC GAP DISAMBIGUATOR

KDD performs static analysis on a kernel's source code to generate a kernel data definition that reflects both direct and indirect relations. KDD also generates a unique signature for each kernel version based on the generated definition to be used in inferring the kernel version, in order to enable systematic mapping of the physical memory. KDD takes the kernel's source code as input and outputs a directed type-graph that represents the kernel data definition. KDD has two main analysis phases to build the type-graph: *direct relations* and *indirect relation analysis*. These steps are discussed below.

A. *Direct Relations Analysis*

This phase generates an initial type-graph that reflects the direct relations between structures that have clear type definitions. KDD performs a compiler-pass approach to extract the data structures (type definitions) by looking for type aliases for *typedef*, and extracts fields within the structures. It then builds an initial type-graph that reflects the direct relations. Nodes are data structures and edges are data members of the structures, as shown in figure 3.

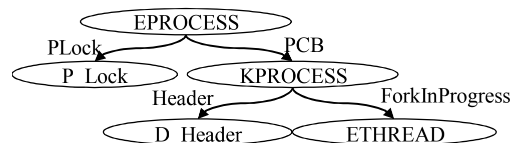

Figure 3. Direct-relations type-graph.

B. *Indirect Relations Analysis*

Indirect inclusion-based relations *e.g.* generic pointer dereferencing cannot be computed from AST directly. To solve this problem, we have developed a new points-to analysis algorithm to statically analyze the kernel's source code, in order to get an approximation for every generic pointer dereferencing based on Anderson's approach. We consider all forms of assignments. Data structures are

flattened on a scalar field. Kernel's objects are represented by their allocation site according to the calling context.

The graph nodes have four types and edges also have four types. ***Nodes;*** represent global and local variables, structures, fields, array elements, procedure argument\ parameters and returns. A node may be: *(i) Variable Node;* represents variables. *(ii) Field Reference Node;* represents structure's fields. *(iii) Function Call Node;* represents a function name and an index; index = -1 if the node represents a function return, otherwise index = *i*, where *i* is the index of formal-in argument. *(iv) Cast Node;* represents explicit casting where the type of the node is the typecast and the name is the casted variable or function. ***Edges;*** directed edges across nodes representing calls, returns and assignments. An edge may be: *(i) points-to edge;* represents points-to relations between two nodes according to the edge direction. *(ii) Inlist edge;* represents a points-to relation between two nodes but on a local scope, thus if ∃ node A has *inlist* edge to node B, then B ∈ *pts(A)* where *pts(A)* means the points-to set of A. *(iii) Outlist edge;* is not a relation edge, but represents a directed path between two nodes that is used to perform the interprocedural and context-sensitive analysis. *(iv) Parent-Child edge;* represents relation between parent and child.

The type-graph of the indirect relations is created and refined by our points-to analysis algorithm in a three step process discussed below.

1) Intraprocedural Analysis

The goal of this analysis is to compute a local type-graph but without information about caller or callees. KDD takes the AST file as input and outputs an initial graph that contains nodes, as follows: *(i)* Variables; create node for each variable declaration and check the function scope to find out if it is a local or global variable. *(i) Procedure definition;* create node for each formal-in parameter. *(ii) Procedure call*; create nodes for each formal-in argument, in addition to a dummy node for each formal-in argument represented by its relative position (index) in the procedure. These dummy nodes will be used later to create an implicit assignment relation between the formal-in arguments and formal-in parameters. For example, given *G(x, y)*, we create two nodes for *x* and *y* and other two dummy nodes *G:1* and *G:2*. *(iii) Assignments;* create nodes for the left and right hand sides. *(iv) Return;* create one node for the return statement itself and one for the returned value.

Meanwhile, KDD builds the initial edges by computing the *transfer function* (TF) as described in table 1. TF is a formal description for the relation between the nodes created for each of the previous entities. In our example, consider the call to *Updatelinks*, where the formal-in parameters are *(src, tgt)*, and the passed arguments are *(ActiveProcessLinks, PsActiveProcessHead)*. *Updatelinks* contains also explicit assignment statements *(src→ Flink = tgt→Flink; tgt→Blink = src→Blink)*. KDD computes the transfer function for those statements as shown in figure 4(a) and 4(b), respectively. For the *return* node, given this fragment of code *UniqueThreadId = ExHandler()*, the computed TF is shown in figure 4(c).

Table 1. Transfer function description.

	Code	Local Points-to Sets
procedure	***Description;*** relation between formal-in parameters and the dummy nodes that hold the indexes of the parameters. ***Edges;*** *inlist* edge between each formal-in parameter node and its relevant dummy node, and *outlist* edge from the dummy node to its relevant formal-in parameter node.	
	proc(p)	*pts (proc:1) ⊇ pts(p)*
Assignment	***Description;*** relation between left and right hand sides of the assignment statement. ***Edges***; *inlist* edge from left hand side to right hand side, and *outlist* edge from the right hand side to left hand side.	
	p=&q	*loc (q) ∈ pts(p)*
	p=q	*pts (p) ⊇ pts(q)*
	*p=*q*	*∀v ∈ pts(q) : pts (p) ⊇ pts(v)*
	p=q*	*∀v ∈ pts(p) : pts (v) ⊇ pts(q)*
Call	***Description;*** relation between the formal-in arguments nodes and dummy nodes. ***Edges***; inlist edge between each argument node and its relevant dummy node.	
	proc(q);	*pts(q) ⊇ pts (proc:1)*
Return	***Description***; relation between left hand side, the procedure return node and the returned value node. ***Edges***; *inlist* edge between the left hand side and the return node, *inlist* edge between return node and retuned value node and *outlist* edge between return node and the left hand side.	
	p = fn() {return q]	*pts (p) ⊇ pts(q)*

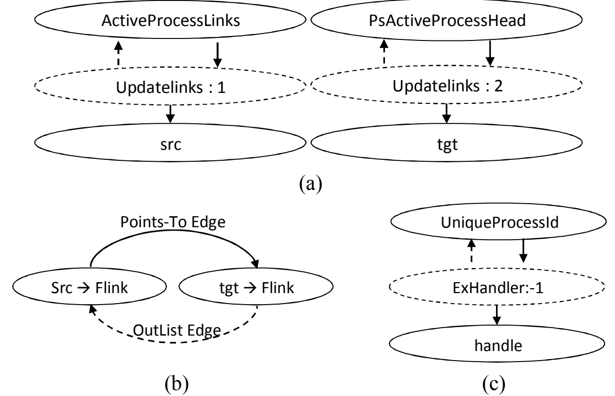

Figure 4. Intraprocedural analysis result graph; solid arrows inlist edges, dashed arrows outlist edges, dashed ovals dummy nodes.

2) Interproceural Analysis

We perform an interprocedural analysis that enables performing the analysis across different files to perform whole-program analysis. We refine the initial type-graph by incorporating interprocedural information from the callees of each procedure. The result of this phase is a graph that computes calling effects (returns, arguments and parameters), but without any calling context information. This is done by propagating the local points-to sets (*inlist* edges) computed at the intraprocedural analysis step to their use sites consistently with argument index in the call site. ∀ N has the form N(Procedure Name : index), we create implicit assignment (*inlist* edge) relation and *outlist* edge between the caller and callee and then delete the dummy node, as shown in figure 5. Thus we could be able to map between procedure arguments and parameters.

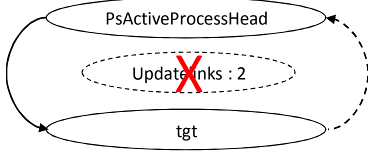

Figure 5. Interprocedural analysis.

3) Context-Sensitive Points-To Analysis

The key in achieving context-sensitivity is to obtain the return of procedures according to the given arguments combined with the call site. This step is performed in three sub-steps discussed below.

a) Points-to Analysis. We build a Procedure Dependency Graph (PDG). This enhances the analysis by providing the appropriate analysis sequence that results in precise points-to analysis. We start with the top node that does not have any dependencies, and thus we guarantee that each node got its *inlist* nodes already analyzed before proceeding with the node itself. We expand the local dereferencing of the pointers to get the points-to relations between the caller and callee. We propagate the points-to set of each node into its successors accumulating to the bottom node. For the acyclic points-to relations, pointers are analyzed iteratively until their points-to sets are fully traversed. For recursions, we analyze pointers in each recursion cycle individually.

b) Graph Unification. Consider this line of code from our example *Updatelinks(&ptr->ActiveProcessLinks, &PsActiveProcessHead)*. We pass an object type to the procedure; however the *Updatelinks* procedure manipulates the object's fields *e.g. Flink* and *Blink*. To solve this problem, we apply a unification algorithm, as follows: given node A with points-to set S and $T \in S$, if T has *child-relation* edge with f; we copy f to A, create a *child-relation* edge between f and A, and also create points-to edge from $A.f$ to $T.f$, as shown in *f*igure 6.

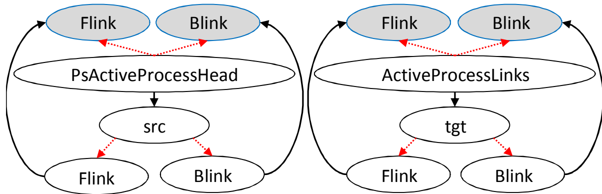

Figure 6. Graph unification: highlighted nodes are the newly copied children nodes. Red arrow shows child-relation edge.

c) Context-Sensitivity. To achieve context-sensitivity, we used the computed transfer function for each procedure and apply its calling contexts, to bind the output of the function call according to the calling site. The points-to edge here is a tuple $\langle n, v, c \rangle$ represents a pointer n points to variable v at context c, where the context is defined by a sequence of functions and their call-sites to find out valid call paths between nodes. Performing context-sensitive analysis solves two problems: the calling context and the indirect (implicit) relations between nodes. These indirect relations are calculated for each two nodes that are in the same function scope but not included in one points-to relation. Such that, $\forall$ two nodes v and n where $v \in pts(n)$, and v and n has different function scope, check the function scope of n and x where $x \in pts(v)$, if the function scope is the same then create a *points-to* edge between n and x. Figure 7 shows the final context-sensitive analysis for the *Updatelinks* example. We find an indirect *points-to* relation between *PsActiveProcessHead* and *ActiveProcessLinks*.

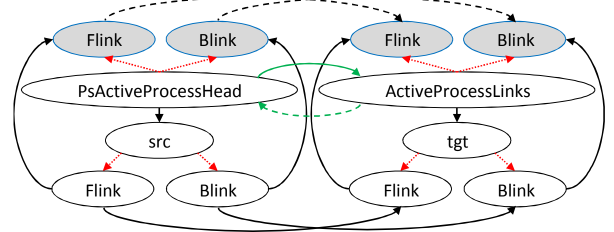

Figure 7. Context-Sensitive Analysis.

C. Kernel Version Checker

To solve the semantic gap efficiently, we need to know the exact kernel version of the running OS kernel where such detailed information is not available for CPs. We used the generated kernel data definition to create a unique signature for each kernel build to infer the kernel version systematically. To do this, we need to pick a data structure that has a different signature for each kernel build, and also it should be a robust structure (structure that should present all the time in the kernel execution). From our observations in Windows [2] (XP SP2 and SP3) and Linux (v3.0.22 and v 3.1.10), we found that no structure is distinctive across the different kernel builds; however the offsets of the members change in each build. On the other hand, Brendan *et al.* [26] found that process structure with some specific fields in it should exist during system runtime and modifying their values will crash the structure. Based on that, we generate a signature for each kernel build. This signature contains the process structure (EPROCESS in Windows and task_struct in Linux) with specific data members (that had been discussed in [26]) combined with their offsets. At runtime, we pick the first loaded process in the processes DL (which is the system process that presents all the time in the kernel execution and termination causes system crash).

IV. Mapping Physical Memory

SVA uses our generated type-graph (TG) to systematically overcome the semantic gap for the VMs hosted in an IaaS platform. SVA is a virtual appliance that utilizes VMI to externally, extract all the high-level information of VM's OS by mapping the hardware bytes to useful high-level information. A high-level representation of this analysis process is shown in Figure 8. Whenever a hosted VM is powered on, SVA is notified by the hypervisor. SVA then creates a separate thread for each VM

[2] We used Windows debugging tools to find the data members' offsets in the different kernel versions, as there is no source code for those versions.

(to enable protecting multiple concurrent VMs using one instance of SVA) using the thread pool manager. SVA first checks the control registers of the VM's processor to get the memory layout of the VM's hardware (there are four main paging modes supported by the hardware that are controlled by the control registers CR0 and CR4), and also performs the kernel version inference check to load the appropriate TG. SVA then starts solving the semantic gap through the Semantic Gap Builder (SGB) by traversing the memory starting from the OS global variables (global variables have static physical addresses in any OS) and then following pointer dereferencing until it covers all memory objects, based on KDD's type-graph. For the Windows, we can get the global variables addresses from Microsoft Symbols [27] and for Linux, the addresses can be obtained from the kernel symbol table file. As VMs including SVA does not have direct access to the server's physical memory, we use the hypervisor (using VMI APIs) to read these physical memory pages into the Memory Pages Buffer (MPB). SVA then installs memory access or timer-based triggers on the memory page(s) that needs to be monitored/ protected according to the applied defence mechanisms (Defence Modules) using the memory access handler (MAH). Whenever a memory access to such pages occurs, the hypervisor (*via* VMI APIs) notifies the MAH, and the hypervisor suspends execution. MAH then loads the requested memory page(s) to the Defence Modules or the SGB to extract kernel data structure updates.

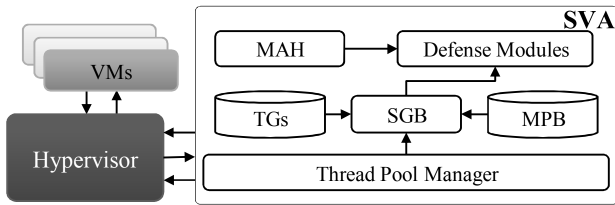

Figure 8. SVA High-level Architecture.

V. IMPLEMENTATION AND EVALUATION

We implemented KDD using C# and a modified version of *pycparser* [28]. KDD uses *pycparser* to generate the AST files of the kernel's source code. As *pycparser* cannot process C directive statements, we developed a C preprocessing tool that solves the directives problem. The preprocessing tool: *(i)* replaces the *#include* with the entire contents of the requested file. *(ii)* Replaces *#define* with any occurrence of the identifier in the rest of the code by the replacement value. KDD starts by the preprocessing tool that takes the kernel's source code as input and outputs a processed C files. *Pycparser* then generates AST files and KDD applies our points-to analysis algorithm on those files to generate the type-graph.

Our implementation and evaluation platform for KDD is 2.2GH core i5 processor with 12 GB RAM. KDD scales to the very large size of such OSs. Table 2 shows the amount of type definitions (data structures/object types), global variables and generic pointers used in the Linux kernel (~ 6 million LOC) and WRK (~ 3.5 million LOC) that have been analysed by KDD. KDD needed 46 hours to analyze the WRK and 72 hours to analyze the Linux kernel. As our analysis was performed offline and just once or each kernel version, the performance overhead of analyzing kernels was acceptable and would not present any problem for any security application that can use KDD. We tried to use a commercial points-to analysis tool, CodeSurfer [29] (the only points-to analysis tool in the market that provides field and context sensitive analysis) to analyze kernels to compare results and performance. However, CodeSurfer could not perform the analysis as it ran out of memory after several days of operation.

To evaluate the effectiveness of KDD results, we performed a comparison between the pointer-based relations inferred by KDD and the manual efforts of security experts to solve these indirect relations in both Linux kernel v3.0.22 and WRK. We manually compared 74 generic pointer structure/ global variable from WRK and 65 from Linux kernel. Table 3 shows the results for few structures (space limits) showing that KDD successfully concluded the candidate target type for them with 100% soundness. KDD is *sound* if the points-to set for each variable contains all its actual runtime targets, and is *imprecise* if the inferred set is larger than necessary. Imprecise results could be sound *e.g.* if $pts(p) = \{a,c,b\}$ while the actual runtime targets are *a* and *b*, then KDD is sound but not precise. KDD is 100% sound as it performs points-to analysis on all program variables not just declared pointers, in order to cover all runtime targets whilst omitting unnecessary local variables. Because of the huge size of the kernel, we could not measure the precision for nearly 60% of the members we used in our experiment, where there is no description for these pointer-based members in the manual efforts. We measured the precision for the well-known objects that had been analyzed manually to be around 96% in both Linux kernel and WRK.

To test the effectiveness of our kernel version inference approach, we used the Windows debugging tools [27] (to get offsets for the different kernel builds) combined with our generated type-graph (to solve the pointer relations) to generate unique kernel signatures for Windows XP SP2, SP3 and 64bit using the EPROCESS structure with the data members discussed in [26]. We succeeded in identifying the kernel version for three memory images for SP2, SP3 and 64bit using our kernel version inference approach. We implemented a prototype of SVA using the VMsafe APIs (specifically vCompute APIs) on VMware ESX hypervisor.

Table 2. Kernel source code initial analysis. 1st column shows the number of type definitions; 2nd column presents the number of global variables and DL column shows the number of doubly linked lists. AST column shows AST files size in gigabyte.

	TD	GV	Void *	Null *	DL	AST
Linux	11249	24857	5424	6157	8327	1.6
WRK	4747	1858	1691	2345	1316	0.9

Table 3. Comparison results between the output of KDD and some facts about the kernel data indirect relations for both Linux and WRK.

OS	Structure / GV	Computed Points-to Sets
Linux	*thread_group* (structure)	task_struct.thread_group:[task_struct.group_leader.thread_group; thread_group.next: [list_head.next, task_struct.thread_group.next, task_struct.group_leader.thread_group]; thread_group.next: [list_head.next, task_struct.thread_group.next, task_struct.group_leader.thread_group] - Context: Thread
	journal_info (void*)	journal_info:[btrfs_trans_handle, gfs2_trans, nilfs_transaction_info]
	cg_list (structure)	cg_list: [list_head, css_set.tasks, css_set __rcu.task] - Context: task_struct
	btrace_seq (void*)	blktrace_seq, unsigned int
Windows	*PsActiveProcessHead* (global variable)	*PsActiveProcessHead:* [List_Entry, ActiveProcessLinks] *PsActiveProcessHead.Flink*: [ActiveProcessLinks.Flink, ActivePr cessLinks.Flink], *PsActiveProcessHead.Blink*: [ActiveProcessLinks.Blink, ActiveProcessLinks.Blink] - Context: EPROCESS
	ThreadListHead (structure)	*ThreadListHead:* [List_Entry], *ThreadListHead.Flink:* [List_Entry.Flink], *ThreadListHead.Blink:* [List_Entry.Blink] - Context: ETHREAD
	LdtInformation (void*)	*LdtInformation*: [PVOID, PROCESS_LDT_INFORMATION]
	DirectoryTableBase (unsigned integer)	*DirectoryTableBase*: [MmCreateProcessAddressSpace:-1], *DirectoryTableBase[0]*: [PageDirectoryIndex, ULONG64], *DirectoryTableBase[1]*: [HyperSpaceIndex, ULONG64]

Our evaluation and implementation platform for SVA is a 2.8 GHz Intel Xeon with 6GB RAM, running ESX 4.1. Figure 9 shows the deployment model of SVA. ESX server hosts SVA and two VMs. SVA is configured with 2GB RAM and deployed as a virtual appliance, running Ubuntu Linux 8.04 Server JeOS, and hosts the vCompute APIs and our code. Our code is normal Linux C program written using vCompute and Posix Threads APIs. SVA is isolated from other server network workloads in a separate virtual network using a virtual switch (vSwitch). Hosted VMs are running Windows XP 64-bit and 32-bit. The objective of this experiment is to prove the effectiveness of KDD and SVA in solving the semantic gap accurately and systematically with sound points-to sets for pointer-based relations, not to detect threats where we utilize a traditional memory traversal technique that is vulnerable to object hiding attacks. SVA traversed the physical memory of the two hosted VM's based on: *(i)* the generated type-graph from KDD, to solve pointer relations, and *(ii)* Microsoft Symbols, to get members offsets for the two kernel versions (as WRK is the only available source code for Windows OSs). For mapping VMs that are running Linux OS, we just need our generated type-graph only to map the physical memory (because we have the source code of each kernel version to be analysed by KDD).

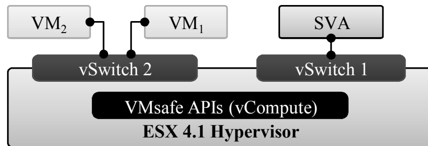

Figure 9. SVA Deployment model.

SVA correctly constructed all the running instances of the data structures at a given memory snapshot using our generated type-graph. The performance overhead of SVA to map the physical memory (all the running objects) was around 12.5 minutes for a memory image of ~ 4GB. To validate that SVA bridged the semantic gap accurately, we compared the external view of mapping the physical memory to runtime objects using SVA with the internal view of the VM using the Windows debugging tools. We compared 43 different object types with their instances. We started from the global variable *PsActiveProcessHead* then followed pointer dereferencing until we covered 43 different objects with their running instances. SVA successfully mapped the correctly identified objects with a low rate of false positives (around 1.5% in traversing balanced trees).

VI. DISCUSSION

Our experiments have shown that SVA using KDD is able to solve the semantic gap for any C-based OS efficiently and systematically. Performing static analysis for the kernel source code to extract robust type definitions for the kernel data has several advantages not just limited to solving the semantic gap: *(i) Systematic Security*; enables implementing systematic security solutions that are able to systematically protect the overall kernel data without the need to understand the kernel data layout. *(ii) Performance Overhead*; minimize the performance overhead in any further security module, where a major part of the analysis process is done offline. If no static analysis were done, every pointer dereferencing would have to be instrumented, which increase the performance overhead. *(iii) Zero-Day Threats;* maximize the likelihood of detecting zero-day threats that target generic or obscure data structures.

To the best of our knowledge, there is no similar research in the area of systematically solving the semantic gap, however KOP [30] has an initiative in systematically computing a type-graph for the kernel data, but KOP is limited in: *(i)* it uses medium-level intermediate representation (MIR) which complicates the analysis and results in improper points-to sets. MIR is extremely big in size, omits very important information such as declarations, data types and type casting, and creates a lot of temporary variables that are allocated identically to source code variables and thus are not easily distinguishable from source code variables *(ii)* the points-to sets of KOP is not highly precise and sound compared to KDD, as KOP depends on the Heintze points-to analysis algorithm [12] which is used in compilers for fast aliasing. *(iii)* KOP could not solve the type ambiguities for DLs. Compared to KOP (used 32GB RAM); KDD has improved performance by around 40%.

Performance also could be improved by increasing the used RAM and processing capabilities. To the best of our knowledge, KDD is the only tool that can scale to produce a detailed, highly accurate type-graph for a large-scale C program such as an OS kernel. This scalability and high performance was achieved by using AST as the basis for points-to analysis. The compact and syntax-free AST improves the time and memory usage efficiency of the analysis. Instrumenting AST is more efficient than instrumenting the machine code (MIR or low-level intermediate representation) because many intermediate computations are saved from hashing.

SUMMARY

The complexity of kernel data makes it impractical to use manual methods to solve the semantic gap problem. In this paper, we presented a new approach that provides a systematic and efficient solution of the semantic gap problem for any C-based OS, without any prior knowledge with the OS. Our experiments showed that KDD efficiently disambiguates the pointer-based relations including generic pointers with high rate of soundness and precision, and enables building an accurate type-graph that reflects the direct and indirect relations of kernel data. SVA, based on the generated type-graph, actively overcomes the semantic gap and reconstructs the running dynamic kernel objects in the VM's physical memory.

ACKNOWLEDGEMENT

Funding provided for this research by the FRST SPPI project and Swinburne University of Technology is gratefully acknowledged. We also thank Swinburne University of Technology for their scholarship support for the first and fourth authors.

REFERENCES

[1] 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.

[2] Common Vulnerabilities and Exposures, "XEN : Security Vulnerabilities " Accessed: Sep 2011, Available: www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html.

[3] Common Vulnerabilities and Exposures, "ESX : Security Vulnerabilities " Accessed: Sep 2011, Available: www.cvedetails.com/vulnerability-list/vendor_id-252/product_id-14181/Vmware-ESX.html.

[4] T. Garfinkel and M. Rosenblum, "Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. of 2003 NDSS*, 2003, pp. 191-206.

[5] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Proc. of 2011 IEEE S&P*, 2011, pp. 297-312.

[6] K. Nance, M. Bishop, and B. Hay, "Virtual Machine Introspection: Observation or Interference?," *J. of IEEE S&P,* pp. 32-37, 2008.

[7] B. D. Payne, M. Carbone, M. Sharif, *et al.*, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Proc. of IEEE Symposium on S&P*, Oakland, CA, 2008, pp. 233-247.

[8] M. Mock, D. C. Atkinson, C. Chambers, *et al.*, "Program Slicing with Dynamic Points-To Sets," *IEEE Trans. Softw. Eng.,* vol. 31, pp. 657-678, 2005.

[9] L. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD Thesis, Copenhagen University, 1994.

[10] B. Steensgaard, "Points-to analysis in almost linear time," in *Proc. of 23rd POPL*, Florida, United States, 1996, pp. 32-41.

[11] D. J. Pearce, P. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis for C," in *Proc. of 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Washington DC, USA, 2004, pp. 37-42.

[12] 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 PLDI*, Utah, USA, 2001, pp. 254-263.

[13] 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 PLDI*, USA, 2007, pp. 278-289.

[14] F. Baiardi, D. Maggiari, and D. Sgandurra, "PsycoTrace: Virtual and Transparent Monitoring of a Process Self," in *Proc. of17th PDP*, Weimar, 2009, pp. 393-397.

[15] 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.

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

[17] 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 CCS*, Virginia, USA, 2007, pp. 128-138.

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

[19] 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.

[20] 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.

[21] 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.

[22] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *Proc. of 2007 ACM SIGPLAN PLDI*, California, USA, 2007, pp. 290-299.

[23] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang, "Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code," in *Proc. of 8th annual IEEE/ACM CGO*, Ontario, Canada, 2010, pp. 218-229.

[24] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *Proc. of 36th POPL*, GA, USA, 2009, pp. 226-238.

[25] R. Ghiya, D. Lavery, and D. Sehr, "On the importance of points-to analysis and other memory disambiguation methods for C programs," in *Proc. of ACM SIGPLAN 2001 PLDI*, Utah, US, 2001, pp. 47-58.

[26] B. Dolan-Gavitt, A. Srivastava, *et al.* "Robust signatures for kernel data structures," in *Proc. of16th CCS*, USA, 2009, pp. 566-577.

[27] Microsoft, "Debugging Tools For Windows," Accessed Dec 2010, Available: msdn.microsoft.com/en-us/windows/hardware/gg463009.

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

[29] GrammaTech. (Accessed: Nov 2011, www.grammatech.com/products/codesurfer/overview.html). *CodeSurfer®*.

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