In Proceedings of the 2012 International Conference on Network and System Security (NSS 2012), Fujian, China, Nov 21-23 2012, LNCS, Springer.

Operating System Kernel Data Disambiguation to Support Security Analysis

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

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

Abstract. It is very challenging to verify the integrity of Operating System (OS) kernel data because of its complex layout. In this paper, we address the problem of systematically generating an accurate kernel data definition for OSes without any prior knowledge of the OS kernel data. This definition accurately reflects the kernel data layout by resolving the pointer-based relations ambiguities between kernel data, in order to support systemic kernel data integrity checking. We generate this definition by performing static points-to analysis on the kernel's source code. We have designed a new points-to analysis algorithm and have implemented a prototype of our system. We have performed several experiments with real-world applications and OSes to prove the scalability and effectiveness of our approach for OS security applications.

Keywords: Points-to Analysis, Operating System, Kernel Data Structures.

1 Introduction

Kernel data rootkits have the ability to alter the overall behavior of Operating Systems (OSes) – without injecting any malicious code into the kernel address space – by manipulating the pointer-based relations between kernel data structures. It is challenging to verify the integrity of kernel data, where an OS kernel contains thousands of data structures that have direct and indirect relations between each other with no explicit integrity constraints. In Windows and Linux OSes, 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 (*e.g.* null pointers that do not have values, and void pointers that do not have associated type declarations in the source code). Such generic pointers get their values or type definitions only at runtime according to the different calling contexts. This makes kernel data a rich target for potent rootkits that exploit the points-to relations between data structures instances to hide/modify system runtime objects *e.g.* processes and threads.

Checking the integrity of OS's kernel data has been a major concern in many OS kernel security researches. However, current research efforts and practices [1-3] are severely limited as they depend on their prior knowledge of kernel data layout to manually resolve the ambiguous pointer-based relations, and thus they only cover 28% of kernel data structures (as discussed by Carbone *et al.* [4]) that relate to well-known objects. Current approaches also do not consider the generic pointer relations between structures, making their approach imprecise and vulnerable to wide range of attacks that can exploit these pointers. This results in security holes, limited protection and an inability to detect zero-day threats. It also raises the need to obtain an accurate kernel data definition that resolves pointer-based relation ambiguities. Such a definition is an important step in implementing different systematic OS security applications *e.g.* memory forensics tools, virtual machine introspection (VMI), dynamic object uncovering and kernel integrity checking tools.

In this paper, we address the problem of systematically building an accurate kernel data definition that precisely models data structures, reflects both direct and indirect relations, and generates constraint sets between structures. We extend KDD (Kernel Data Disambiguator), a tool that can generate a sound kernel data definition for any C-based OS (*e.g.* Windows, Linux, UNIX) without any prior knowledge of the OS kernel data layout [19]. KDD disambiguates pointer-based relations including generic pointers – to infer their candidate types/values – by performing static *points-to* analysis on the kernel's source code. *Points-to* analysis is the problem of determining statically a set of locations to which a given variable may point to at runtime. *Points-to* analysis for C programs has been widely used in compiler optimization, memory error detection and program understanding [5,6]. However, none of these approaches meet our requirements in analyzing the kernel as they do not scale to the enormous size and complexity typical of an OS kernel. They also typically sacrifice precision for performance. In KDD, precision is an important factor. We want the most precise points-to sets to be computed. To meet our requirements, we implemented a new *points-to* analysis algorithm that has the ability to provide inter-procedural, context-sensitive, field-sensitive and inclusion-based *points-to* analysis for large programs that contain millions lines of code *e.g.* OS kernel. We have implemented a prototype system of KDD, and performed several experiments on KDD using large-scale real-world applications including OSes to prove its effectiveness and scalability.

Section 2 presents the motivation for our work and key related work. Section 3 presents KDD's approach. We discuss implementation and evaluation details in Section 4. Finally we discuss results and draw key conclusions.

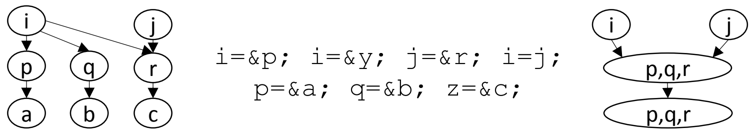
2 Background

Ensuring reliability of large systems *e.g.* OSes is a difficult problem, especially C-based ones. C-based OSes use C structures heavily to model objects. They also use pointers extensively to simulate call-by-reference semantics, emulate object-oriented dispatch *via* function pointers, avoid expensive copying of large objects, implement lists, trees and other complex data structures, and also as references to objects allocated dynamically on the heap [7]. Moreover, objects can be cast to multiple types during their lifetime, and a pointer deposited in a field under one object may be read from a field under another object. This makes the analysis of kernel's data structures a nontrivial task, further complicated by the fact that data structures are implementation-dependent. Hence, imprecise points-to analysis will therefore result in improper assumptions about the indirect relations between structures. As C allows casting, values can be copied from a pointer to a non-pointer and vice versa, points-to sets should be computed to all program variables, not just declared pointers.

To get a concrete idea of the pointers problem in OSes, we discuss three problems of generic pointers we need to address. **First**, use of void pointers; void pointers support a form of polymorphism. At runtime, if a pointer is not void, its target object should have the type of a pointer. However, the problem with use of 'void' type is that the target object type(s) can only be identified at runtime. The wide use of such void pointers hinders performing systematic integrity checks on kernel data, where there are no type constraints for void *. This assists hackers in exploiting these pointers to point to somewhere else in memory. **Second**, use of null pointers; these are used for example to implement linked lists (e.g. single, doubly or triply) which are heavily used in OSes to maintain running objects. The C definition makes a linked list points to a linked list, but actually during runtime it points to a specific object type according to the calling contexts. The problem is that the objects structured in a list can be recognized only during runtime (*e.g.* object type, number of running instances and locations). Thus, null pointers manipulation helps hackers to hide or change runtime objects. Identifying the object type that a list may hold at the offline analysis phase helps significantly in identifying a set of constraints on the runtime objects to detect invalid pointer manipulations. **Third**, use of casting; C data types can be subverted by casting. A pointer of a given type can be cast to point to any other C type. A major problem with casts is that they induce relationships between objects that appear to be unrelated, enabling hackers to exploit data structure layout in physical memory.

Kernel data integrity checking has been studied intensively [8,9,2,1]. However, these research efforts are limited to the OS expert knowledge to resolve the ambiguous pointer relations. OSck [8] and SigGraph [10] follow a systematic approach to resolve the pointer-based relation, however they do not solve the generic pointers (indirect relations) problem. To the best of our knowledge, KOP [4] (a Microsoft internal tool) , is the first and only tool that employed points-to analysis in order to analyze an OS kernel to solve generic pointers ambiguities. However, KOP is limited. It uses a medium-level intermediate representation (MIR) that 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 [11]. Also in KOP, the points-to sets of the void * objects are not precise and thus they use a set of constraint criteria (OS-specific) at runtime to find out the appropriate candidate for the object.

Many state-of-the-art tools have been developed for points-to analysis of C programs [5,6,12]. Their use has predominantly been for compiler optimizations and program understanding, and their main goal has thus been performance. They differ mainly in how they group alias information. There are two main algorithms used to group alias information: *Andersen's* [13] and *Steensgaard's* [14]. Fig. 1 shows a C code fragment and the points-to sets computed by those algorithms. Andersen's is the slowest but the most precise while Steensgaard's is the fastest but is imprecise. Anderson's approach creates a node for each variable and the node may have different edges. Steensgaard's algorithm groups alias sets in one node and each node has one edge. Based on these approaches there are different types of analysis that trade-off performance and precision: *(i) Field-Sensitivity*; distinguishing the different fields inside structures and unions *i.e.* each field has a distinct points-to set. *(ii) Context-Sensitivity*; distinguishing objects created through different call sites. Context-sensitive algorithms are more precise, but are much slower in performance and complicated to implement. *(iii) Flow-Sensitivity;* considers the effects of pointer assignments with respect to the call-graph. *(iv) Inclusion-Based;* considers dependency relations between structures to represent the inclusion constraints.



```
i=&p; i=&y; j=&r; i=j;
  p=&a; q=&b; z=&c;
```

Fig. 1. Alias information grouping by Steensgaard's and Andersen's approaches.

A number of research efforts have attempted performing field and context sensitivity analysis on large programs [15,6,16,17]. However none have been shown to scale to large programs *e.g.* OS's kernel code with a high rate of precision. Also, these algorithms are used during program compilation time to name objects by allocation site, not by access path. Thus, they do not allow us to disambiguate null pointers.

3 Our Approach

KDD takes the source code of an OS kernel as input and outputs an accurate directed type-graph that represents the kernel data definition. This type-graph 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. A high-level representation of this analysis process is shown in Fig. 2. To facilitate the analysis, we use Abstract Syntax Tree (AST) as a high-level intermediate representation for the source code. The AST captures the essential structure of the code that reflects its semantic structure while omitting unnecessary syntactic details. Since it has been established that flow-sensitivity does not add significant precision over a flow-insensitive when we ignore the control-flow of programs [18], we consider flow-insensitive points-to analysis in KDD.

Two main phases of the analysis are used to build the type-graph. The first analysis step is straightforward, and its goal is computing the direct relations between kernel data structures that have clear type definitions. This is done by performing a compiler-pass approach on the AST files to extract the data structure type definitions by looking for *typedef* aliases, and extract their fields with the corresponding type definition. Nodes are data structures and edges are data members of the structures. The second step is the most important step and its goal is computing the indirect relations between structures. Indirect relations (generic pointer dereferencing) cannot be computed from the AST directly. To solve this problem, we have developed a new points-to analysis algorithm to statically analyze the kernel's source code to get an approximation for every generic pointer dereferencing based on Anderson's approach. We consider all forms of assignments and function calls. Data structures are flattened to a scalar field. Type casting is handled by inferring locations accessed by the pointer being cast. Kernel objects are represented by their allocation site according to the calling contexts. The target of this step is a graph *G (N, E)*, where *N* is the set of nodes representing global and local variables, fields, array elements, procedure arguments\parameters and function returns. *E* is a set of directed edges across nodes representing, assignments and function calls. The graph has different types of nodes and edges (details omitted for brevity - for more details please see [19]). The type-graph of this step is created and refined by our points-to analysis algorithm in three steps, discussed below.

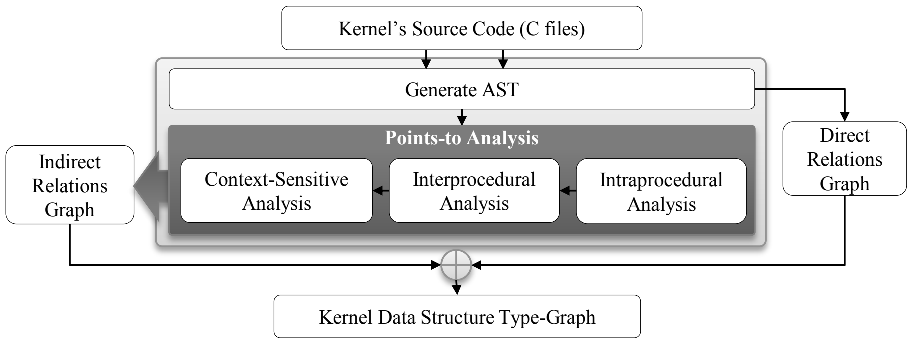


Fig. 2. High-level view of KDD operation.

3.1 Intraprocedural Analysis

The goal of this analysis step is to compute a local type-graph but without information about caller or callees. Algorithm 1 summarizes our intraprocedural analysis algorithm. 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. *(ii) Procedure definition;* create node for each formal-in parameter. *(iii) 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*. *(iv) Assignments;* create nodes for the left and right hand sides. *(v) Function 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.

Algorithm 1 Intraprocedural Analysis Algorithm

```
1:  Procedure IntraproceduralAnalysis (ASTFile F)
2:          ∀ ASTLine L ∈ F
3:          if L ∋ variable V declaration statement then check function scope;
4:              if (scope == null) then V ⊆ global variable
5:              elseif L ∋ function parameters then V ⊆ Local function parameter
6:              else V ⊆ Local variable
7:              Create node();
8:          endif
9:          if L ∋ assignment | function call | return statement then Compute transfer function ();
10: end
```

Table 1. Transfer function description; local points-to sets *pts()*, constraints between nodes, and edges (→ a directed *inlist* edge between two nodes, ← a directed *outlist* edge).

	Code	Local pts()	Constraints	Edges
Proc	***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)*	*proc:1 ⊇ p*	*proc:1 → p, proc:1 ← p*
Assignment	***Description;*** relation between left and right hand sides (HSs) of the assignment statement. ***Edges***; *inlist* edge from left HS to right HS, and *outlist* edge from the right HS to left HS.			
	p=&q	*loc (q) ∈ pts(p)*	*p ⊇ [q]*	*p → q, p ← q*
	p=q	*pts (p) ⊇ pts(q)*	*p ⊇ q*	*p → q, p ← q*
	*p=*q*	*∀ v ∈ pts(q) : pts (p) ⊇ pts(v)*	*p ⊇ *q*	*p → *q → v, p ← *q ← v*
	**p=q*	*∀ v ∈ pts(p) : pts (v) ⊇ pts(q)*	**p ⊇ q*	*v → *p → q, v ← *p ← 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)*	*q ⊇ proc:1*	*q → proc:1*
Return	***Description***; relation among 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 the return node and retuned value node and *outlist* edge between the return node and the left hand side.			
	p = fn() return q;	*pts (p) ⊇ pts(q)*	*p ⊇ q*	*p → q*

Consider a call to a procedure called "*Updatelinks"*, where the formal-in parameters are *(src, tgt)*, and the actual passed arguments are (&ActiveProcessLinks, &ActiveProcessHead), and consider these explicit assignment statements *(src→Flink = tgt→Flink; tgt→Blink = src→Blink)*. KDD computes the TF for those statements as shown in Fig. 3(a) and Fig. 3(b), respectively. For the *return*, given this fragment of code UniqueThreadId = ExHandler(), the computed TF is shown in Fig. 3(c).

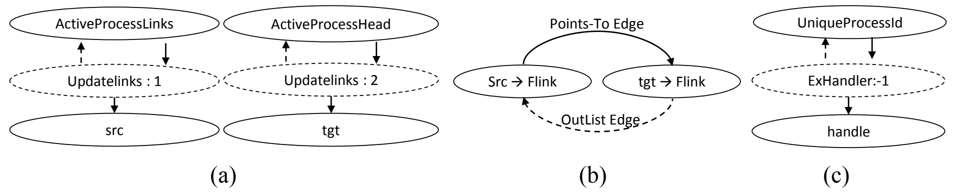
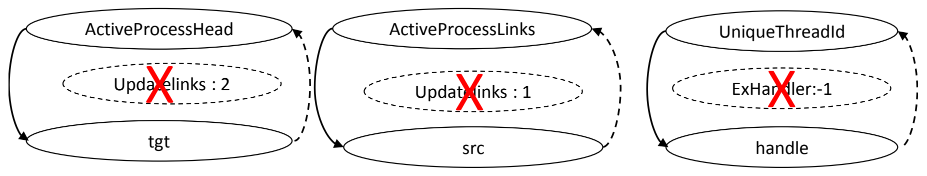


Fig. 3. Intraprocedural analysis graph.

3.2 Interprocedural Analysis.

In this phase we perform an interprocedural analysis that enables performing points-to analysis across different files to perform whole-program analysis. The result of this phase of analysis is a graph that computes the calling effects (returns, arguments and parameters), but without any calling context information yet. This is done by propagating the local points-to sets computed at the intraprocedural step to their use sites consistently with argument index in the call site. Thus we can map between the procedure arguments and parameters. Fig. 4 shows the analysis results of this step for the examples discussed in the intraprocedural analysis step. Algorithm 2 summarizes this interprocedural analysis step.



(a) (b)

Fig. 4. Interprocedural analysis result.

Algorithm 2 Intraprocedural Analysis Algorithm

```
1: Procedure Interprocedural Analysis (Graph G)
2:     ∀ Node N ∈ G
3:         if ∃ N has the form N(Procedure Name : index) then
4:             Create inlist edge (N.outlist, N.inlist); Create outlist edge (N.intlist, N.outlist);
6:             Delete dummy nodes ();
7:         end if
8: end
```

3.3 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. Algorithm 3 summarizes our context-sensitive analysis of this step, performed in three sub-steps, as follows:

Points-to Analysis. A well-known complication in this analysis is the order of which nodes will be analyzed first, where this can greatly affect performance. A good choice is to analyze nodes in a topological order [12], by building a Procedure Dependency Graph (PDG). Our PDG consists of nodes representing the statements of the data dependency in the program. Data dependency between two statement nodes exists if a variable at one statement might reach the usage of the other variable at another statement. We start with the top node (according to the computed PDG) that does not have any dependencies, and thus we guarantee that each node has 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. Then 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 to make the analysis algorithm accommodates to modification and read effects introduced by the calls.

Algorithm 3 Points-to Analysis

```
1:  Procedure PointsToAnalysis (PDG PDG, Graph G, TransferFunction TF)
2:      ∀ Node N ∈ G
3:          ∀ InListNode in ∈ N.InList
4:              Compute points-to set (in);   N. PointstoSet. Add (in. PointstoSet);
5:          N. PointstoSet. Add (in);
6:          ∀ PointedToNode toN ∈ N. PointstoSet
7:              ∀ Child ch ∈ N. Children
8:                  CopyNode (ch);   Connect edges ();
9:          UpdateNodePointsTo (N, toN); Write the Graph();
10: end procedure
11: Procedure UpdateNodePointsTo (Node N, PointedToNode toN)
12:     if N.fnScope != toN.fnscope) then ∀ SubPointedToNode StoN ∈ toN. PointstoSet
13:         if StoN.fnScope == N.fnScope then N. PointstoSet. Add(StoN);
14:     else UpdateNodePointsTo (N, toN);
15: end procedure
```

Graph Unification. This step targets to compute a consistent graph. Consider the following procedure and procedure call: `void updatelinks(PList_Entry src, PList_Entry tgt)` and `Updatelinks(&ptr->ActiveProcessLinks, &ActiveProcessHead)`. We pass an object type to the procedure; however *Updatelinks* manipulates the fields of passed object *Flink* and *Blink*. As the definition of the _PList_Entry data type is: typedef struct _LIST_ENTRY { struct _List_Entry *Flink; struct _List_Entry *Blink;} List_Entry, *PList_Entry;

To solve this problem, we apply a unification algorithm to the type-graph, as follows: given node A with points-to set S and $T \in S$, if T has *child-relation* edge with f; we create a *points-to* edge between f and A. Fig. 5(a) shows the analysis result of this step of this example piece of code.

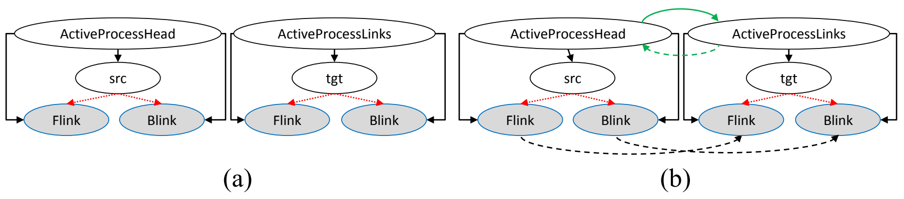


Fig. 5. Context-Sensitive Analysis.

Context-Sensitivity. Without context-sensitivity, the analysis of functions that have different calling context would result in very general points-to sets for their arguments. To achieve context-sensitivity, we used the transfer function for each procedure call and apply its calling contexts, to bind the output of the procedure 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 set. Such that, ∀ 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*. Fig. 5(b) shows the final context-sensitive analysis for the *Updatelinks* example. We discovered that there is an indirect *points-to* relation from *ActiveProcessHead* to *ActiveProcessLinks*.

Finally, we write the type-graph. We replace each variable node with its data type and for fields and array elements we add the declared parent type. Finally, we format the results of our analysis to the DOT language, as a simple visualization for the kernel data layout to be used by the other OS security solutions that make use of KDD.

4 Implementation and Evaluation

We have implemented a prototype of KDD using C#. KDD uses *pycparser* [20] to generate AST files of the kernel's source code. KDD then uses the AST files to applying 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). In the intraprocedural analysis, KDD analyzes each AST file using a separate thread. For interprocedural analysis, KDD allocates a thread for each procedure to parses the AST files to map between the procedure parameters and arguments. However, for the context-sensitive, the analysis is done on sequential-basis as each node depends on its predecessors.

We performed three types of experiments with KDD to demonstrate its scalability and effectiveness. We measured the soundness and precision of KDD using different sets of benchmarks. We analyzed the Linux kernel v3.0.22 and WRK ((Windows Research Kernel) using KDD, and performed a comparison between the computed pointer relations using KDD, and the manual efforts to solve these relations in both kernels. We implemented a memory mapping tool that uses our type-graph to correctly map the physical memory bytes from an introspected virtual machine to actual runtime objects, in order to prove KDD efficiency in defining the kernel data. Our implementation and evaluation platform is 2.5GHz core i5 processor with 12 GB RAM.

4.1 Soundness and Precision

The points-to analysis algorithm 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 the algorithm is sound but not precise and thus there exist false positives. If $pts(p) = \{a,c\}$ and the actual runtime targets are *a* and *b* then the algorithm is not sound nor precise, and thus there exist false positives and negatives. KDD is sound as it performs the points-to analysis on all program variables not just declared pointers, in order to cover all runtime targets whilst omitting unnecessary local variables.

We used a selection of C programs from the SPEC2000 and SPEC2006 benchmark suites, and other open source C programs, to measure the soundness and precision of KDD. Table 2 shows the characteristics of these benchmark C programs, in addition to the precision of the KDD analysis of each. We also show indications of memory, time and processor usage of running KDD on these benchmark programs. We manually verified each program to get an accurate estimation of the points-to set. For programs that are less than 4 KLOC, we instrumented pointers manually. For larger programs we picked a random set of generic pointers based on our understanding of the program. However, we could not measure precision for some programs because of their size. We also ran each program and monitored allocations in physical memory to get the actual runtime targets (i.e. relevant points-to set). Then, we used the equation below to calculate precision. Our results show that for the benchmark C programs analyzed by KDD, we achieved a high level of precision and 100% of soundness. The results also show that for significantly sized C programs KDD is able to process the application code with very acceptable CPU time and memory usage.

$$Precision = \frac{Relevant\ Points\ to\ Set\ \cap\ Retrieved\ Points\ to\ Set}{Retrieved\ Points\ to\ Set}$$

Table 2. Soundness and Precision Results running KDD on a suite of benchmark C programs. LOC is lines of code. Pointer Inst is number of pointer instructions. Proc is number of Procedure definitions. Struct is number of C struct type definitions. AST T is time consumed to generate the AST files, AST M is memory usage, and AST C is CPU usage. TG T is time consumed to build the type-graph, TG M is memory usage, TG C is CPU usage.

Benchmark	LOC (K)	Pointer Inst	Proc	Struct	ASTt (sec)	ASTm (MB)	ASTc (%)	TGt (sec)	TGm (MB)	TGc (%)	P (%)
art	1.2	286	43	19	22.7	21.5	19.9	73.3	12.3	17.6	100
equake	1.5	485	40	15	27.5	25.4	20.4	87.5	14.1	21.1	98.6
mcf	2.4	453	42	22	43.2	41	28.5	14	23	27	97.2
gzip	8.6	991	90	340	154.2	144.6	70.5	503.3	81.4	68.3	95.1
parser	11.3	3872	356	145	305.2	191.2	76.7	661.4	107.8	74.3	94.5
vpr	17.7	4592	228	398	316.1	298.7	80.2	1031.5	163.2	79	NA
gcc	222.1	98384	1829	2806	3960.5	3756.5	93.5	12962	2200	94	NA
sendmail	113.2	9424	1005	901	2017.2	1915.1	91.6	6609	1075.0	91.5	NA
bzip2	4.6	759	90	14	82.3	78.1	45.5	271.6	44.2	42.9	95.9

4.2 Kernel Analysis

To illustrate the scale of the problem presented by C-based OSes, we performed a simple statistical analysis on the WRK (~ 3.5 million LOC) and Linux kernel v3.0.22 (~ 6 million LOC) to compute the amount of type definitions (data structures), global variables and generic * used in their source code. Table 3 summarizes this analysis.

Table 3. Kernel source code analysis.
1st column shows the number of type definitions, 2nd column is the number of global variables, DL column shows the number of doubly linked lists and last column reflects the number of unsigned integers that represent casting problem. AST column shows AST files size in gigabyte.

	TD	GV	Void*	Null*	DL	Uint	AST
Linux	11249	24857	5424	6157	8327	4571	1.6
WRK	4747	1858	1691	2345	1316	2587	0.9

KDD scales to the very large size of such OSes. KDD needed around 28 hours to analyze the WRK and around 46 hours to analysis the Linux kernel. Comparing KDD to KOP, KOP has to be run on a machine with 32 GB RAM and needed around 48 hours to analyze the Windows Vista kernel. The performance of KDD could be improved by increasing RAM and processing capabilities. As our points-to analysis is performed offline and just once for each kernel version, the performance overhead of analyzing kernels is acceptable. It does not present a problem for any security application that wants to make use of KDD's generated type graph. Re-generation of the graph is only necessary for different versions of a kernel where data structure layout changes may have occurred. Security tools can use the KDD-generated type graph for the particular version of an OS to which they are applied.

To evaluate the accuracy of KDD's generated OS type graphs, we performed a comparison between the pointer relations inferred by KDD and the manual efforts of OS experts to solve these indirect relations in both kernels. We manually compared around 74 generic pointers from WRK and 65 from the Linux kernel. These comparisons show that KDD successfully deduced the candidate target type/value of these members with 100% soundness. Because of the huge size of the kernel, we could not measure its precision for nearly 60% of the members we used in our experiment, as there is no clear description for these members from any existing manual analysis. We were thus only able to measure precision for well-known objects that have been analyzed manually by security experts and whose purpose and function is well-known and documented. The resulting precision was around 96% in both kernel versions.

4.3 Object-Graph for Security Monitoring

We modified our earlier-developed kernel security monitoring tool, *CloudSec* [21], to use our KDD-generated type-graph to traverse the physical memory of a running OS from a hypervisor level, in order to construct a correct object-graph that identifies all the running instances of the data structures for a running Virtual Machine (VM). The objective of this experiment was to demonstrate the effectiveness of KDD in computing a precise kernel data definition, not to detect threats where we utilize a traditional memory traversal technique that is vulnerable to object hiding attacks. *CloudSec* is a security appliance that has the ability to monitor VMs' memory from outside the VM itself, without installing any security code inside the VM. *CloudSec* uses memory traversal techniques to map the running objects based on manual profiles that describe the direct and indirect relations between structures [21]. In this experiment, we used our KDD-generated type-graph to locate dynamic objects by traversing kernel memory starting from the OS global variables and then following pointer dereferencing until we covered all memory objects. We used *CloudSec* to map the physical memory of a VM running Windows XP 64bit. The performance overhead of *CloudSec* to construct the object-graph for the entire kernel running objects was around 6.3 minutes for a memory image of 4GB on a 2.8 GHz CPU with 6GB RAM. To evaluate the mapping results, we considered the global variable *PsActiveProcessHead* then followed pointer dereferencing until we covered 43 different data structures with their running instances. We compared the results with the internal OS view using Windows Debugger. *CloudSec* successfully mapped and correctly identified the running kernel objects, with a low rate of false positives (~1.5% in traversing balanced trees). This demonstrates that, for these 43 data structures monitored, our generated type-graph is accurate enough for kernel data disambiguation to support security monitoring.

5 Discussion

KDD is a static analysis tool that operates offline on an OS kernel's source code to generate a robust type-graph for the kernel data that reflects both the direct and indirect relations between structures, models data structures and generates constraint sets on the relations between them. Our experiments with KDD have shown that the generated type-graph is accurate, and solves the null and void pointer problems with a high percentage of soundness and precision. KDD is able to scale to the enormous size of kernel code, unlike many other points-to analysis tools. This scalability and high performance was achieved by using an AST as the basis for points-to analysis. The compact and syntax-free AST improves time and memory usage efficiency of the analysis. Instrumenting the AST is more efficient than instrumenting the machine code *e.g.* MIR because many intermediate computations are saved from hashing.

Performing static analysis on kernel source code to extract robust type definitions for the kernel data structures has several advantages: *(i) Systematic Security*; enables the implementation of systematic security solutions. By this we mean that we have the ability to systematically protect kernel data without the need to understand deep details about kernel data layout in memory, as is done to date. *(ii) Performance Overhead;* we minimize 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. *(iii) Detecting Zero-Day Threats;* we maximize the likelihood of detecting zero-day threats that target generic (*via* bad pointer dereferencing) or obscure kernel data structures. *(iv) Generating Robust Data Structures Signatures*; KDD generates robust data structure signatures that can be used by brute force scanning tools [10]. *(v) Type-Inference;* declared types of C variables are unreliable indications of how the variables are likely to be used. Type inference determines the actual type of objects by analyzing the usage of those objects in the code base. *(vi) Function Pointer Checking*; enable checking the integrity of kernel code function pointers that reside in dynamic kernel objects, by inferring the target candidate type for each function pointer. This decreases the need to instrument every function pointer during runtime, as the addresses of objects that hold these pointers change during runtime.

To the best of our knowledge, there is no similar research in the area of systematically defining the kernel data structure with the exception of KOP [4]. However in addition to the limitations discussed in the related work section, the points-to sets of KOP are not highly precise compared to KDD. This is because they depend on the Heintze points-to analysis algorithm [6], which is used in compilers for fast aliasing. In addition, KOP computes transitive closures in order to perform the points-to analysis - this increases the performance overhead of the analysis. To the best of our knowledge, our points-to analysis algorithm is the first points-to analysis technique that depends on an AST to provide interprocedural, context and field sensitive analysis. Buss *et al.* [22] has an initiative in performing points-to analysis based on the AST of the source code. However, their algorithm is field and context insensitive.

6 Summary

The wide existence of generic pointers in OS kernels makes kernel data layout ambiguous and thus hinders current kernel data integrity research from providing the preemptive protection. In this paper, we described KDD, a new tool that generates a sound kernel data structure definition for any C-based OS, without prior knowledge of the OS kernel data layout. Our experiments with our prototype have shown that the generated type-graph is accurate and solves the generic pointer problem with a high rate of soundness and precision. Extension of our CloudSec security appliance for external VM data structure checking shows it can assist OS security applications.

7 Acknowledgement

The authors are grateful to Swinburne University of Technology and FRST Software Process and Product Improvement project for support for this research.

8 References

1. Baliga, A., Ganapathy, V., Iftode, L.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: Proc of 2008 Annual Computer Security Applications Conference 2008, pp. 77-86. IEEE Computer Society, 1468197
2. Dolan-Gavitt, B., Srivastava, A., Traynor, P., Giffin, J.: Robust signatures for kernel data structures. In: Proc. of 16th ACM conference on Computer and communications security, Illinois, USA 2009, pp. 566-577. ACM, 1653730
3. Ibrahim, A., Shouman, M., Faheem, H.: Surviving cyber warfare with a hybrid multiagent-base intrusion prevention system. IEEE Potentials **29**(1), 32-40 (2010).
4. Carbone, M., Cui, W., Lu, L., Lee, W.: 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. ACM, 1653729
5. Whaley, J., Lam, M.S.: 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. ACM, 996859
6. Heintze, N., Tardieu, O.: 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. ACM, 378855
7. Mock, M., Atkinson, D.C., Chambers, C., Eggers, S.J.: Program Slicing with Dynamic Points-To Sets. IEEE Trans. Softw. Eng. **31**(8), 657-678 (2005). doi:10.1109/tse.2005.94

8. Hofmann, O.S., Dunn, A.M., Kim, S.: 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. ACM, 1950398
9. Petroni, N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proc of 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA 2007, pp. 103-115. ACM, 1315260
10. Lin, Z., Rhee, J., Zhang, X.: 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
11. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.H.: Oblivious Hashing: A Stealthy Software Integrity Verification Primitive. In: Proc. of 5th International Workshop on Information Hiding 2003, pp. 400-414. Springer-Verlag
12. Pearce, D.J., Kelly, P.H., Hankin, C.: 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. ACM, 996835
13. Andersen, L.: Program Analysis and Specialization for the C Programming Language. University of Copenhagen (1994)
14. Steensgaard, B.: Points-to analysis in almost linear time. In: Proc. of 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Florida, United States 1996, pp. 32-41. ACM, 237727
15. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Proc. of 2007 ACM SIGPLAN conference on Programming language design and implementation, California, USA 2007, pp. 290-299. ACM, 1250767
16. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proc. of 8th annual IEEE/ACM international symposium on Code generation and optimization, Ontario, Canada 2010, pp. 218-229. ACM, 1772985
17. Lattner, C., Lenharth, A., Adve, V.: 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.
18. Hind, M., Pioli, A.: Which pointer analysis should I use? In: Proc of 2000 ACM SIGSOFT international symposium on Software testing and analysis, Portland, Oregon, United States 2000, pp. 113-123. ACM, 348916
19. Ibrahim, A.S., Grundy, J.C., Hamlyn-Harris, J., Almorsy, M.: Supporting Operating System Kernel Data Disambiguation using Points-to Analysis. In: Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany 2012
20. Bendersky, E.: pycparser: C parser and AST generator written in Python (2011, Available at http://code.google.com/p/pycparser/).
21. Ibrahim, A.S., Hamlyn-Harris, J., Grundy, J., Almorsy, M.: 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.
22. Buss, M., Edwards, S.A., Bin, Y., Waddington, D.: Pointer analysis for source-to-source transformations. In: Proc. of 5th IEEE International Workshop on Source Code Analysis and Manipulation, 30 Sept.-1 Oct. 2005 2005, pp. 139-148