Supporting Operating System Kernel Data Disambiguation using Points-to 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

Generic pointers scattered around operating system (OS) kernels make the kernel data layout ambiguous. This limits current kernel integrity checking research to covering a small fraction of kernel data. Hence, there is a great need to obtain an accurate kernel data definition that resolves generic pointer ambiguities, in order to formulate a set of constraints between structures to support precise integrity checking. In this paper, we present KDD, a new tool for systematically generating a sound kernel data definition for any C-based OS *e.g.* Windows and Linux, without any prior knowledge of the kernel data layout. KDD performs static points-to analysis on the kernel's source code to infer the appropriate candidate types for generic pointers. We implemented a prototype of KDD and evaluated it to prove its scalability and effectiveness.

Categories and Subject Descriptors

D2.7 [**LOGICS AND MEANINGS OF PROGRAMS**]: *Program analysis*; D4.6 [**Operating Systems**]: *Security kernels.*

General Terms

Performance, Security, Languages.

Keywords

Systematic kernel data integrity checking, points-to analysis.

1. INTRODUCTION

It is a very challenging task to verify the integrity of OS kernel data. An OS kernel has thousands of data structures that have direct and indirect relations between each other, with no explicit integrity constraints. In Windows and Linux OSs, from our analysis, nearly 40% of the 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 and thus types only at runtime according to the calling context. This makes kernel data a rich target for malware that exploits the pointer relations between data structures to compromise the kernel. Current kernel data integrity checking research [1, 2] is limited in solving those problems. This is because they depend on their prior knowledge of the kernel data to manually resolve the ambiguous pointer-based relations, and thus they only cover a small fraction of kernel data structures that relate to well-known objects *e.g.* processes and threads [3]. This results in limited protection and inability to detect zero-day threats, raising the need to get an accurate kernel data definition that resolves the generic pointers ambiguities.

In this paper, we introduce KDD (Kernel Data Disambiguator), a new static analysis tool that can generate a sound kernel data definition for any C-based OS (*e.g.* Windows and Linux), without any prior knowledge of the OS kernel data layout. KDD disambiguates the pointer relations including generic pointers - to infer their candidate types/values - by performing static points-to analysis on the kernel's source code. KDD is able to scale to the enormous size of kernel code, unlike many other points-to analysis tools. 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. To meet our requirements, we designed and implemented a new points-to analysis algorithm that has the ability to provide interprocedural, context- and field-sensitive, and inclusion-based points-to analysis for large programs that contain millions lines of code *e.g.* OS kernel.

2. BACKGROUND

C-based OSs use C structures heavily to model objects. They also use pointers extensively to emulate object-oriented dispatch, avoid expensive copying of large objects, implement complex data structures. 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 non-trivial task. To get a concrete idea of the generic pointers problem, Figure 1 shows exemplar C code implementing pointers of the sort found in a typical OS. We discuss in it the context of three problems we need to address: *void pointers*, *null pointers* and *casting*.

Void pointers; the problem with use of 'void' type is that the target object type(s) can only be identified at runtime. From our example, *UniqueProcessId* is void *. However if we analyse the code, we find that it indirectly points to another data structure, *_ExHandle*. The wide use of such void pointers hinders performing systematic integrity checks on kernel data, where there are no type constraints for void *. ***Null Pointers;*** null pointers are used heavily to implement linked lists which are heavily used in OS kernels. The C definition makes a linked list 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 used to update the objects' list structured in *_LIST_ENTRY* (doubly-linked list). However, the objects structured in this list can be recognized only during runtime. Identifying type of the object that a linked list may hold at the offline analysis phase helps significantly in identifying a set of constraints on the runtime objects to detect invalid pointer dereferencing. ***Casting;*** a major problem with casts is that they induce relationships between objects that may appear to be unrelated, enabling hackers to exploit data structures layout in physical memory. *DebugPort*, from our code, is declared as an integer; however it is being cast to be a pointer to a data structure.

2.1 Related Work

Pointer analysis algorithms for C programs have been studied intensively over the last two decades. 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 [4, 5]. However none has been shown to scale to large programs *e.g.* OS's kernel code with a high precision rate. Kernel data integrity checking has been studied intensively [1, 2]. However, all of these approaches depend on OS expert knowledge to extract some value-invariants that cover specific semantics for a very small number of kernel data structures. OSck [2] and SigGraph [6] provides a more systematic approach to cover system data, however they do no not solve generic pointers problem.

```
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 reflecting use of generic pointers.

3. OUR APPROCH

KDD takes an OS kernel's source code as input and outputs a type-graph that summarizes the different data types located in the kernel along with their connectivity patterns. It reflects inclusion-based relation between kernel data structures – for both direct and indirect relations – to generate constraint sets between data structures. To facilitate the analysis, we use Abstract Syntax Tree (AST) as a high-level intermediate representation for the source code. KDD proceeds by first generating the AST for the kernel's source code. Then two main phases of the analysis are used to build the type-graph: *(i) Direct Inclusion-Based Relations;* to extract kernel type definitions to build an initial type-graph that reflects the direct relations between structures; *(ii) Indirect Inclusion-Based Relations;* to compute the indirect relations.

3.1 Direct Inclusion-Based Relations

This phase of analysis is straightforward, and its output is an initial type-graph that reflects the direct inclusion-based relations between kernel data structures that have clear type definitions. From the generated AST file, KDD performs a compiler-pass approach 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 (inclusion relations) of the structures

3.2 Indirect Inclusion-Based Relations

Indirect relations (generic pointer dereferencing) cannot be computed from the AST directly. We have developed a new points-to analysis algorithm to statically analyse the kernel's source code to get an approximation for every generic pointer dereferencing based on Anderson's approach [5]. 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 context. The target graph of this step is *G (N, E)*, where *N* is the set of nodes representing global and local variables, fields, array elements, procedure arguments\parameters and function return. *E* is a set of directed edges across nodes representing, assignments and function calls. The graph nodes have four types and edges also have four types. ***Nodes*** - a node is one of: *(i) Variable Node;* represents variables including parameters. *(ii) Field Reference Node;* represents structure's fields. Each field reference node has an associated parent node. *(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 – *i.e.* given a function call *G (arg1, arg2)* in this case we will have two nodes *G:1* and *G:2* representing passed arguments *arg1* and *arg2*, respectively. *(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*** - 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 are used to achieve the points-to analysis. *(iv) Parent-child edge;* represents relation between parent and child – *i.e.* relation between structure and fields, or array and elements.

The type-graph of the indirect relations is created and refined by our points-to analysis algorithm in a three step process: *Intraprocedural Analysis*, *Interprocedural Analysis*, and *Context-Sensitive Points-To Analysis*. These steps are discussed below.

3.2.1 Intraprocedural Analysis

The goal of this phase is to compute a local type-graph without information about caller or callee. KDD takes the AST file as input and outputs an initial graph, as follows: *(i) Variables* **-** create a node for each variable declaration and check the function scope to find out if it is a local or global variable. *(ii) Procedure Declaration;* create a node for each formal-in parameter; *(iii) Call;* create node for each formal-in argument (if not already created), in addition to a dummy node for each formal-in argument represented by its index in the procedure. These dummy nodes will be used in the interprocedural analysis phase to create an implicit assignment relation between the formal-in arguments and formal-in parameters. *(iv) Assignments;* create nodes for the left and right hand sides, if not already created. *(iv) Returns;* create two nodes; one for the return statement itself and the other for the returned value inside the called procedure.

KDD then builds the initial edges at this step by computing a *transfer function* (TF) for each procedure, procedure call, assignment, and return statement, 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 motivating example from Section 2, consider the call to the function *Updatelinks*, where the formal-in parameters are *(src, tgt)*, and the actual passed arguments are *(&ActiveProcessLinks, &PsActiveProcessHead). Updatelinks also* contains explicit assignment statements *(src→ Flink = tgt→Flink; tgt→Blink = src→Blink).* KDD computes the transfer function (TF) for those statements as shown in Figure 2 (a) and Figure 2 (b), respectively. For the *return* node, given this fragment of code *UniqueThreadId = ExHandler()*, the computed TF is shown in Figure 2 (c).

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
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)*	*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*

3.2.2 Interprocedural Analysis

In this phase we perform an interprocedural analysis that enables us to perform points-to analysis across different files to perform a 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 the calling effects (returns, arguments and parameters), but without any calling context information yet. 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, as shown in figure 3. Thus we can map between the procedure arguments and parameters.

3.2.3 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. Points-to analysis algorithm of this step, performed in three sub-steps as follows:

1) ***Points-to Analysis***; a well-known complication in this analysis is the order of which nodes will be analysed first, where this can greatly affect performance. A good choice is to analyse nodes in a topological order [7], by building a Procedure Dependency Graph (PDG). This graph enhances the analysis by providing the appropriate analysis sequence that result in precise points-to analysis. We start with the top node that does not have any dependencies, and thus we guarantee that each node has its *inlist* nodes already analysed 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 acyclic points-to relations, pointers are analysed iteratively until points-to sets are fully traversed. For recursions, we analyse pointers in each recursion cycle individually.

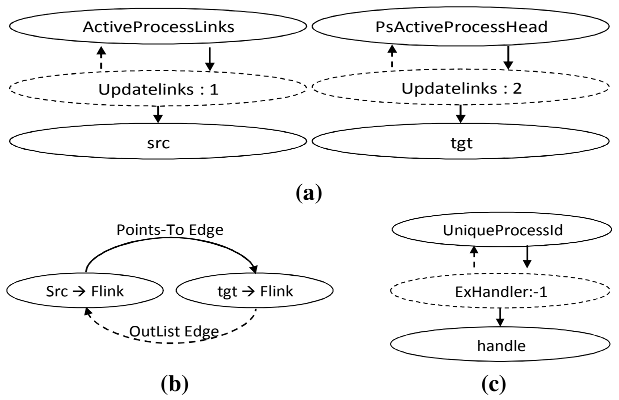

Figure 2. Intraprocedural analysis graph: solid arrows inlist edges and dashed outlist edges; dashed ovals dummy nodes.

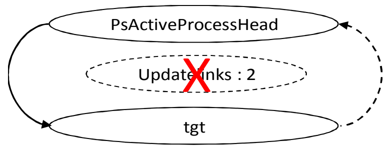

Figure 3. Interprocedural analysis result.

2) ***Graph Unification;*** consider the following piece of code from the motivating C code example: *Updatelinks (&ptr->ActiveProcessLinks, &PsActiveProcessHead).* We pass an object (data structure) to the procedure; however the procedure *Updatelinks* manipulates the fields of the passed object *e.g. Flink* and *Blink.* 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 ∈ 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 Figure 4.

Context-Sensitivity; to achieve context-sensitivity, we use 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 ⟨*n*, *v*, *c*⟩ representing that 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 of the 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* ∈ *pts(n)* and *v* and *n* has different function scope, check the function scope of *n* and *x* where *x* ∈ *pts(v)*, if the function scope is the same then create a *points-to* edge between *n* and *x*. Figure 5 shows the final context-sensitive analysis for *Updatelinks*. Note an indirect *points-to* relation from *PsActiveProcessHead* 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.

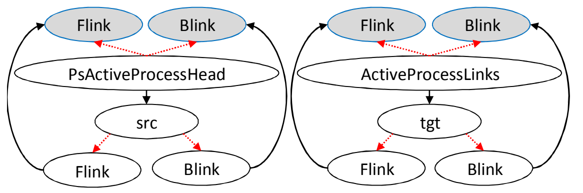

Figure 4. Graph unification: highlighted nodes are the newly copied children nodes. Red arrow shows child-relation edge.

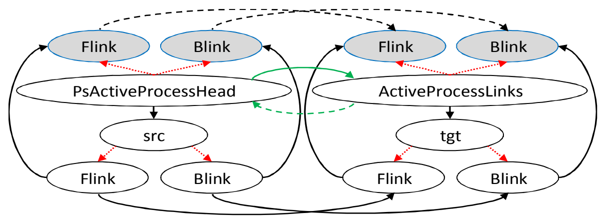

Figure 5. Context-Sensitive Analysis.

4. EVALUATION

To demonstrate KDD's scalability and effectiveness we first measured the soundness and precision of KDD using different sets of C programs from the SPEC2000 and SPEC2006 benchmark suites and other open source programs. Our results, shown in Table 2, show that we achieve a high level of precision (~ 97%) and 100% of soundness. For significantly sized C programs KDD is able to process the application code with very acceptable CPU time and memory usage. Second, we analyzed the Linux kernel v3.0.22 (~ 6 million LOC) and WRK (~ 3.5 million LOC). KDD scales to the very large size of such OSs. KDD needed around 46 hours to analyze the WRK and around 72 hours to analysis the Linux kernel. As our points-to analysis is performed offline and just once or each kernel version, performance overhead of analyzing kernels is acceptable and does not present a problem for any security application that wants to make use of KDD's generated type graph. To evaluate the effectiveness of KDD results, 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. KDD successfully deduced the candidate target type/value of these members with 100% soundness. We could not measure the precision for nearly 60% of the members as there is no clear description for these members from any existing manual analysis. We measured precision for well-known objects and precision was around 96%.

Thus KDD is able to scale to produce a detailed, highly accurate type-graph for a large-scale C program such as an OS kernel. A key to achieve this scalability and high performance was by using AST as the basis for points-to analysis. The compact and syntax-free AST improves time and memory usage efficiency of the analysis. 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 [3]. However, KOP is limited in that the points-to sets of KOP are not highly precise; analysis performance overhead is very high; and KOP uses a medium-level intermediate representation (MIR) which complicates the analysis and results in improper points-to sets.

5. SUMMARY

The wide existence of generic pointers in OS kernels makes kernel data ambiguous and thus hinders current kernel data integrity research from providing the preemptive protection. KDD is a new tool that has the ability to generate a sound kernel data structure definition for any C-based operating system, without any prior knowledge of the OS. Our experiments have shown that the KDD-generated type-graph is accurate and solves the generic pointer problem with high rate of soundness and precision. To the best of our knowledge, KDD is the only tool that can scale to produce a detailed, highly accurate type-graph for C-based OSs.

ACKNOWLEDGEMENTS

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

6. REFERENCES

[1] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *Proc of 2008 Annual Comp. Sec. App. Conf.*, 2008, pp. 77-86.

[2] O. S. Hofmann, A. M. Dunn, and S. Kim, "Ensuring operating system kernel integrity with OSck," in *Proc. of 16th ASPLOS*, California, USA, 2011, pp. 279-290.

[3] M. Carbone, W. Cui, L. Lu, *et al.* "Mapping kernel objects to enable systematic integrity checking," in *Proc 16th ACM CCS*, 2009, pp. 555-565.

[4] 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.

[5] L. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD, U. of Copenhagen, 1994.

[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 NDSS*, San Diego, 2011.

[7] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis for C," in *Proc. of 5th ACM SIGPLAN-SIGSOFT* PASTE, USA, 2004, pp. 37-42.

Table 2. Soundness and Precision Results running KDD on a suite of benchmark C programs.

LOC lines of code. Pointer Inst number of pointer instructions. Proc number of Procedure definitions. Struct number of C structs AST T time consumed to generate AST files, AST M memory usage, and AST C CPU usage. TG T time consumed to build the type-graph, TG M memory usage, TG C CPU usage.

Benchmark	LOC	Pointer Inst	Proc	Struct	AST T (sec)	AST M (MB)	AST C (%)	TG T (sec)	TG M (MB)	TG C (%)	P (%)	S (%)
art	1272	286	43	19	22.7	21.5	19.9	73.3	12.3	17.6	100	100
equake	1515	485	40	15	27.5	25.4	20.4	87.5	14.1	21.1	98.6	100
mcf	2414	453	42	22	43.2	41	28.5	14	23	27	97.2	100
gzip	8618	991	90	340	154.2	144.6	70.5	503.3	81.4	68.3	95.1	100
parser	11394	3872	356	145	305.2	191.2	76.7	661.4	107.8	74.3	94.5	100
vpr	17731	4592	228	398	316.1	298.7	80.2	1031.5	163.2	79	NA	100
gcc	222185	98384	1829	2806	3960.5	3756.5	93.5	12962	2200	94	NA	100
sendmail	113264	9424	1005	901	2017.2	1915.1	91.6	6609	1075.0	91.5	NA	100
bzip2	4650	759	90	14	82.3	78.1	45.5	271.6	44.2	42.9	95.9	100