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
+61 (3) 9214 8731
[aibrahim, jgrundy, jhamlynharris, malmorsy]@swin.edu.au

ABSTRACT
It is very challenging to verify the integrity of Operating System (OS) kernel data. An OS kernel contains thousands of data structures that have direct and indirect relations between each other, with no explicit integrity constraints. Nearly 40% of these relations are pointer (indirect) relations, and 35% of these pointer relations are generic pointers. Generic pointers make kernel data layout ambiguous, and thus limit kernel integrity checking research to covering a small fraction of kernel data. There is a great need to obtain an accurate kernel data definition that resolves pointer relations 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 OS. KDD generates this definition by performing static points-to analysis on the 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 provides interprocedural, context-sensitive, field-sensitive and inclusion-based points-to analysis for large C programs. We have implemented a prototype of KDD and performed several experiments to prove its scalability and effectiveness.

Categories and Subject Descriptors
D2.7 [LOGICS AND MEANINGS OF PROGRAMS]: Semantics of Programming Languages – Program analysis; D4.6 [Operating Systems]: Security and Protection – Security kernels.

General Terms

Keywords
Systematic kernel data integrity checking, points-to analysis and large-scale automated software analysis.

1. INTRODUCTION
Kernel data rootkits have the ability to alter the overall behavior of Operating Systems (OSs), and hide or modify runtime objects (e.g., processes, loaded modules, services) without injecting any malicious code into the kernel address space [1-3]. Thus, protecting and checking the integrity of OS’s kernel data has been a major concern in OS kernel security research, for both in-guest (OS-hosted) security solutions [4, 5], and external virtualization-aware security solutions [1, 6]. It is a very challenging task to verify the integrity of kernel data, where an OS kernel has thousands of data structures (data types in the kernel code that reflects the different runtime object types) that have direct and indirect relations between each other with no explicit integrity constraints. In Windows and Linux OSs, from our observations, 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 structures a rich target for malware that exploits the pointer relations between data structures to compromise the kernel. Since data structure syntax is controlled by the OS code, exploiting generic pointers in the kernel memory image will not make the OS to treat the exploited structure as an invalid instance of a given type. Current kernel data integrity checking research and practices [1-3, 7, 8] are severely limited in solving those problems as: (i) they only cover 28% of kernel data structures that relate to well-known objects e.g., processes and threads [9]; (ii) they depend on their prior knowledge of kernel data layout to manually resolve ambiguous pointer-based relations and they thus cover a small fraction of kernel data [7, 8]; and (iii) most current kernel data protection research relies on using the field value-invariant of data structures as a signature [1-3], which is not sufficient to defend against zero-day threats i.e. previously unknown exploits. Many kernel data structures cannot be covered by the value-invariant scheme. For example, it is difficult to generate field-invariants for data structures that are part of linked lists (single, doubly and triply linked lists, all common in OS kernels), because the actual running contents of these structures depend on the calling contexts at runtime. These limitations result in security holes, limited protection and inability to detect zero-day threats. These issues raise the need to get an accurate kernel data definition that reflects the direct relations and resolves the pointer-based relations ambiguities. Such a definition is an important step to implement different systematic OS security applications e.g. memory forensics tools, virtual machine introspection (VMI), dynamic object mapping and kernel data integrity checking.

In this paper, we address the problem of how to systematically build an accurate kernel data definition that precisely models data structures, reflects both direct and indirect relations, and generates constraint sets between structures. We introduce KDD (Kernel Data Disambiguator), a new automated software 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. KDD disambiguates the pointer-based relations including generic pointers - to infer their candidate types/values - by performing static points-to analysis on the kernel’s source code. KDD takes the source code of an OS kernel as input and outputs an accurate directed type-graph that represents the kernel data definition. KDD is able to scale to the enormous size of kernel code, unlike many other points-to analysis tools.
Points-to analysis is the problem of determining statically a set of locations to which a given variable may point to at a particular program point [10]. Points-to analysis for C programs has been widely used in compiler optimization, debugging, memory error detection and program understanding [11-14]. However, none of these approaches meet our requirements in analysing the kernel, as these approaches do not scale to the enormous size and complexity typical of an OS 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 in our analysis. To meet our requirements, we designed and implemented a new points-to analysis algorithm that has the ability to provide interprocedural, context-sensitive, field-sensitive and inclusion-based points-to analysis for large programs that contain millions of lines of code e.g. OS kernel. We have implemented a prototype system for KDD and performed several experiments to prove its effectiveness and scalability. We measured the soundness and precision of KDD using different sets of benchmarks. We analyzed Linux kernel v3.0.22 and Windows Research Kernel (WRK - Windows is a commodity OS and WRK is the only available source code for it. WRK packages core Windows XP x64/Server 2003 kernel. This NT kernel is nearly the same in all Windows versions from Windows 2000 to Windows 7 except Vista) using KDD, and performed a comparison between the computed pointer relations, and the manual efforts to solve these relations in both kernels. We implemented a memory mapping tool that uses the type-graph produced by KDD to map the physical memory bytes to actual runtime objects. We evaluated the performance overhead of KDD in these experiments.

Section 2 presents the motivation for our work: ambiguous data structure implementations within OS kernel code bases and we review key related work. Section 3 presents the approach embodied in KDD. In section 4 the algorithm KDD uses to construct a type-graph from OS kernel C code. We discuss implementation issues in Section 5, in section 6 we evaluate our tool with different experiments, and we finally conclude.

2. BACKGROUND

Ensuring reliability and security of large systems e.g. OSs is a difficult problem, especially C-based ones. C-based OSs 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 [10]. 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, further complicated by the fact that data structures are implementation-dependant. Imprecise points-to analysis will therefore result in improper assumptions about kernel data indirect relations. 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.

2.1 Motivating Example

To get a concrete idea of the pointer disambiguation problem, Figure 1 shows exemplary C code implementing pointers of the sort found in a typical C-based OS kernel. We discuss in this example the context of three exemplar problems we need to address: void pointers, null pointers and casting.
DL point to itself, but actually during system runtime it points to a specific object type (e.g. threads or loaded modules) according to the calling context. Procedure `UpdateLinks`, from our example, is 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 (e.g. object type, number of running instances and locations). Thus, DL manipulation helps hackers dramatically to hide or change runtime objects [27]. Identifying type of the object that a DL may hold at the offline analysis phase helps significantly in identifying a set of constraints on the runtime objects to detect invalid pointer dereferencing or manipulation. Context-sensitive [14, 20, 22]. However, these algorithms are used generic pointers problem). Several pointer analysis algorithms are variables must be a non-null path (which does not solve the transitive closure. This assumes that an edge between two, they did not consider the generic pointers problems of the indirect et al. They also group pointer alias information using Steensgaard’s classifications as upcasts (from a pointer to an object) and downcasts (from object to a pointer) [15]. 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. Casts are also often used in function returns e.g. `AllocateMemory`.

### 2.2 Points-to Analysis

Many state-of-the-art tools have been developed for points-to analysis of C programs [13, 16-18]. They differ in how they group alias information. There are two main algorithms used to group alias information: Andersen’s [17] and Steensgaard’s [18]. 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 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 the heap 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.

### 2.3 Related Work

Pointer analysis algorithms for C programs have been studied intensively over the last two decades [12-14, 16]. 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 [16, 19, 20]. However none has been shown to scale to large programs e.g. OS’s kernel code with a high precision rate. Yu et al. [20] proposed a context and field sensitive pointer analysis algorithm based on the static single assignment (SSA) form. SSA is widely used in compiler optimizations to get the points-to information for variables only, but not for structures. They also group pointer alias information using Steensgaard’s approach that does not meet our analysis requirements. Hardekopf et al. [21] proposed a flow-sensitive pointer analysis approach but they did not consider the generic pointers problems of the indirect dereferencing. Heintze [16] 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 [14, 20, 22]. However, these algorithms are used during program compilation time to name objects by allocation site, not by the access path. Such algorithms do not enable solving the ambiguity of null pointers. Buss et al. [23] presented a pointer analysis tool that performs analysis based on AST, but their tool is field and context insensitive, to perform code optimizations. Almost all of the previous research fails to provide sufficient precision and soundness for OS kernel data disambiguation.

Kernel data integrity checking has been studied intensively [2, 3, 8, 24]. Petroni et al. and [2] Arati et al. [1] verified the kernel data integrity of kernel data structures based on manually developed specifications that cover specific semantics for a very small number of kernel data structures. OScK [8] is a kernel data integrity system that extracts data structure definitions from kernel source code and then generates a set of APIs so that users can write the appropriate integrity constraints. However, OScK does not solve the pointer-based relations. Zhiqiang [25] presented SigGraph; a tool that generates signatures for kernel data structures. However, SigGraph only resolves typed pointers and direct relations between data structures without the ability to solve generic pointers, making their approach unable to generate complete and robust signatures for the kernel. To the best of our knowledge, KOP [9] (a Microsoft internal tool), is the first and only tool that employed points-to analysis in order to analyse OS kernel to solve generic pointers ambiguities. However, KOP is quite limited in that: it uses a 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 [26]. 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. KOP cannot solve type ambiguities for casting and null pointers problems e.g. DL, thus KOP at the static analysis phase cannot decide for example what type of objects a DL will hold at run-time. KOP does not consider function recursions in its points-to analysis. KOP assumes that it has the ability to detect hidden objects based on the traditional memory traversal techniques that are vulnerable to object hiding attacks [27]. Petroni et al. [24] proposed a similar approach that depends on performing points-to analysis on kernel’s source code to check control flow integrity, but their tool is limited to function pointers. This does not help in computing a complete type-graph for the kernel, where such graph requires performing points-to analysis on all pointer-compatible forms including calls, returns, structures, assignments and variables.

### 3. OUR APPROACH

KDD is a new kernel data disambiguation tool has the ability to provide an accurate kernel data definition for any C-based OS e.g. Linux and Windows. KDD performs static analysis on the kernel’s source code to solve type ambiguities for pointer-based relations. KDD takes a kernel’s source code as input and outputs an accurate directed type-graph. This type-graph 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 Figure 2. To facilitate the analysis, we use Abstract Syntax Tree (AST) as a high-level intermediate representation for the source code. The AST captures essential structure of the code that reflects its semantic structure while omitting unnecessary syntactic details. Using AST avoids the cost of building the full transitive closure and SSA that are usually
used in traditional points-to analysis algorithms. 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 [28], we consider flow-insensitive points-to analysis in KDD.

Kernel’s heap objects using malloc are represented by their allocation site according to the calling context.

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; KDD extracts kernel type definitions from the AST files to build an initial type-graph that shows direct relations between structures. (ii) Indirect Inclusion-Based Relations; to compute the indirect relations. We perform interprocedural, context sensitive, field-sensitive and inclusion-based points-to analysis. The output of this step is the final type-graph that reflects indirect relations for the generic pointer typed members. In Section 4 we discuss these two main phases of the analysis in details.

4. TYPE-GRAph SYNTHESIS

The aim of KDD is to build a type-graph, \( G(N, E) \) for kernel data, where \( N \) is the set of nodes representing structures/members, and \( E \) is a set of directed edges across nodes representing inclusion relations. This graph summarizes the different data types located in the kernel along with their connectivity patterns.

4.1 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. Figure 3 shows the generated type-graph.

4.2 Indirect Inclusion-Based Relations

Indirect relations e.g., 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 analyse 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 (direct or indirect). Data structures are flattened to a scalar field. Type casting is handled by inferring locations accessed by the pointer being cast.

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 typcast 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 \( \exists \) node \( A \) has inlist edge to node \( B \), then \( B \in pts(A) \) 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 perform the interprocedural and context-sensitive analysis. (iv) Parent-child edge; represents relation between parent and child – i.e. relation between structure and fields, or array and its 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.

4.2.1 Intraprocedural Analysis

The goal of this phase of analysis is to compute a local type-graph but without information about caller or callee. Algorithm 1 summarizes our intraprocedural analysis algorithm. In this step, 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) Procedure 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 later to create an implicit assignment relation between the formal-in arguments and formal-in parameters (in the interprocedural analysis phase). For example, given \( G(x, y) \), we create two nodes for \( x \) and \( y \) (if not already created) and other two dummy nodes \( G:1 \) and \( G:2 \) that reflect the index of each argument. (iv) Assignments - create nodes for the left and right hand sides, if not already created. (iv) Return instruction - 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,
In our motivating example from Section 2, consider the call to the function $\text{ExHandler}$, where the formal-in parameters are $(\text{src}; \text{tgt})$, and the actual passed arguments are $(\text{src} =$ ExHandler(), and the actual passed arguments are $(\text{ActiveProcessLinks}; \text{PsActiveProcessHead})$. $\text{UpdateLinks}$ also contains explicit assignment statements ($\text{src} \rightarrow \text{Flink} = \text{tgt} \rightarrow \text{Flink}; \text{tgt} \rightarrow \text{Blink} = \text{src} \rightarrow \text{Blink}$). KDD computes the transfer function (TF) for those statements as shown in Figure 4 (a) and Figure 4 (b), respectively. For the return node, given this fragment of code $\text{UniqueThreadid} = \text{ExHandler}()$, the computed TF is shown in Figure 4 (c).

### Algorithm 1: Intraprocedural Analysis Algorithm

1. **Procedure** IntraproceduralAnalysis (ASTFile $F$)
2. \( \forall \text{ASTLine } L \in F \)
3. if $L \in \text{Variable Declaration statement}
4. then check function scope;
5. if (scope == null) then
6. $V \subseteq \text{global variable}
7. else $L \in \text{Function parameters then}
8. $V \subseteq \text{Local function parameter}
9. else $V \subseteq \text{Local variable}
10. Create node;
11. endif
12. if $L \in \text{assignment | function call | return statement then}$
13. Compute transfer function;
14. end

<table>
<thead>
<tr>
<th>Code</th>
<th>Local pts()</th>
<th>Constraints</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>relation between formal-in parameters and the dummy nodes that hold the indexes of the parameters. <strong>Edges:</strong></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>proc($p$)</td>
<td>pts (proc:1) $\geq$ pts($p$)</td>
<td>proc:1 $\geq$ p</td>
<td>proc:1 $\rightarrow$ p</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>relation between left and right hand sides (HSs) of the assignment statement. <strong>Edges:</strong></td>
<td>inlist edge from left HS to right HS, and outlist edge from the right HS to left HS.</td>
<td></td>
</tr>
<tr>
<td>proc($p$)</td>
<td>pts ($p$) $\geq$ pts($q$)</td>
<td>$p$ $\rightarrow$ $q$</td>
<td>$p$ $\rightarrow$ $q$</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>relation between the formal-in arguments nodes and dummy nodes. <strong>Edges:</strong></td>
<td>inlist edge between each argument node and its relevant dummy node.</td>
<td></td>
</tr>
<tr>
<td>proc($q$)</td>
<td>pts($q$) $\geq$ pts (proc:1)</td>
<td>$q$ $\geq$ proc:1</td>
<td>$q$ $\rightarrow$ proc:1</td>
</tr>
<tr>
<td><strong>Return:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p = \text{fn}()$</td>
<td>pts ($p$) $\geq$ pts($q$)</td>
<td>$p$ $\rightarrow$ $q$</td>
<td>$p$ $\rightarrow$ $q$</td>
</tr>
</tbody>
</table>

Table 1. Transfer function description; local points-to sets $\text{pts}()$, constraints between nodes, and edges ($\rightarrow$ a directed inlist edge between two nodes, $\geq$ a directed outlist edge).

4.2.2 Interprocedural Analysis

The output of phase one is an interprocedural-based graph that does not consider relations between different functions. In this phase we perform an interprocedural analysis that enables performing points-to analysis across different files to perform whole-program analysis. We refine the initial type-graph by incorporating inter-procedural information from the callees of each procedure. 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 (inlist edges) computed at the intraprocedural analysis step to their use sites consistently with argument index in the call site, as shown in Figure 5. Thus we can map between the procedure arguments and parameters. Algorithm 2 summarizes this interprocedural analysis step.

### Algorithm 2: Intraprocedural Analysis Algorithm

1. **Procedure** InterproceduralAnalysis (Graph $G$)
2. $\forall$ Node $N \in G$
3. if $N$ has the form N(Procedure Name : index) then
4. Create inlist edge (N.inlist, N.inlist);
5. Create outlist edge (N.outlist, N.outlist);
6. Delete dummy nodes;
7. end if
8. end

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

Figure 5. Interprocedural analysis result.

4.2.3 Context-Sensitive Points-To Analysis

To get the greatest possible precision, we use a context-sensitive 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 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 [13], 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 (according to the computed PDG) 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. 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 analysed iteratively until their points-to sets are fully traversed. For recursions, we analyse 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)
5: N. PointstoSet. Add (in, PointstoSet)
6: N. PointstoSet. Add (in)
7: ∀ PointedToNode toN ∈ N. PointstoSet
8: ∀ Child ch ∈ N. Children
9: CopyNode (ch);
10: Connect edges;
11: UpdateNodePointsTo (N, toN)
12: Write the Graph();
13: end procedure

16: Procedure UpdateNodePointsTo (Node N, PointedToNode toN)
17: if N.fnScope != toN.fnScope) then
18: ∀ SubPointedToNode StoN ∈ toN. PointstoSet
19: if StoN.fnScope == N.fnScope then
20: N. PointstoSet. Add (StoN);
21: end if
22: else
23: UpdateNodePointsTo (N, toN);
24: end procedure

2) **Graph Unification** - consider the following piece of code from the motivating C code example: UpdateLinks(kptr->ActiveProcessLinks, &PsActiveProcessHead). We pass an object type (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 to T. Figure 6 shows the analysis result of this step of this example piece of code.

3) **Context-Sensitivity** - without context-sensitivity, the analysis of functions that have different calling context (e.g. UpdateLinks in our motivating example that updates the doubly linked lists of process and thread data types) 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 (n, v, c) represents a pointer n points to variable v at context c, where the context is a sequence by defined 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 ∈ 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 7 shows the final context-sensitive analysis for the UpdateLinks example. We discovered that there is 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. Table 2 shows the points-to analysis for our motivating example in section 2. We have used a textual representation here to make it more readable.

**5. IMPLEMENTATION**

We have implemented a prototype of KDD using C# and a modified version of pycparser [29]. KDD uses pycparser to generate the AST files of the kernel’s source code. However a problem we encountered with pycparser is that it cannot process directive statements (directives are lines included in the code that are not program statements but directives for preprocessor). We have developed a C preprocessing tool that resolves the directives including constants. The preprocessing tool: (i) replaces the #include directive by the entire contents of the requested file. Directives are executed in the order in which our preprocessor encounters them. (ii) replaces #define by any occurrence of identifier in the rest of the code by the replacement value. Analyser starts by our C preprocessing tool that takes the kernel source code as input and output processed C code files. Then pycparser tool generates the AST files for the entire kernel source code. KDD then uses the AST files to applying our points-to analysis algorithm to generate the type-graph. Finally, we format the results of our analysis to the DOT language [30]. DOT language is a standard graph description language that supports the specification of undirected and directed graphs. This can be fed into other tools e.g. visualization tools to examine the graph. We have used Microsoft’s Parallel Extensions [31] to leverage multicore processors in an efficient and scalable manner to implement KDD.
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 \( \text{pts}(p) = \{a, c, b\} \) while the actual runtime targets are \( a \) and \( b \), then the algorithm is sound but not precise and thus there exist false positives. If \( \text{pts}(p) = \{a, c\} \) 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 some C programs from the SPEC2000 and SPEC2006 benchmark suites, and other open source C programs, to measure the soundness and precision percentage of KDD. Table 3 shows the characteristics of the used benchmarks in addition to their analysis result soundness and precision. We also show indications of memory, time and processor usage of running KDD on these benchmark programs. We used a commercial points-to analysis tool, CodeSurfer [32] (CodeSurfer is the only points-to analysis tool in the market that provides field- and context-sensitive analysis) to get estimation for the points-to sets of each variable in the used benchmark programs to compare results and performance to KDD. However, CodeSurfer results was not quite accurate, thus we manually verified each program to get an accurate estimation (retrieved points-to set). For programs that are less than 4 KLOC, we instrumented pointers manually. For larger programs we instrumented nearly 50% of the pointers. However, we could not measure precision for some programs because of their size. We also ran each program and monitored object allocation in the physical memory to get the actual runtime targets (relevant points-to set). Then, we used the below equation to calculate the precision. These results show that for the benchmark C programs analyzed by KDD, we achieve a high level of precision and 100% of soundness. This means that the resultant type graph accurately identifies the types of C program data structures even when void * pointers and pointer casting are used extensively. 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.

\[
\text{Precision} = \frac{\text{Relevant Points to Set} \times \text{Retrieved Points to Set}}{\text{Retrieved Points to Set}}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Points-to Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEPROCESS</td>
<td>(PEPROCESS)AllocatePrMemory, PEPROCESS</td>
</tr>
<tr>
<td>PEPROCESS.PkProcess.ThreadHeadList</td>
<td>PLIST_ENTRY, ExHandle.handle</td>
</tr>
<tr>
<td>PEPROCESS.ActiveProcessLinks</td>
<td>PLIST_ENTRY, PsActiveProcessHead</td>
</tr>
<tr>
<td>PEPROCESS.ActiveProcessLinks.Blink</td>
<td>PLIST_ENTRY.Blink, PEPROCESS.ActiveProcessLinks.Blink, PsActiveProcessHead.Blink</td>
</tr>
<tr>
<td>PEPROCESS.ActiveProcessLinks.Flink</td>
<td>PLIST_ENTRY.Flink, PEPROCESS.ActiveProcessLinks.Flink, PsActiveProcessHead.Flink</td>
</tr>
<tr>
<td>PSActiveProcessHead</td>
<td>PEPROCESS.PkProcess.ThreadHeadList, PLIST_ENTRY</td>
</tr>
<tr>
<td>PSActiveProcessHead.Flink</td>
<td>PLIST_ENTRY.Flink, PEPROCESS.PkProcess.ThreadHeadList.Flink, PsActiveProcessHead.Flink</td>
</tr>
<tr>
<td>ExHandler. ExHandle</td>
<td>ExHandle.handle, ExThread.CreateHandler.</td>
</tr>
<tr>
<td>ActiveProcess</td>
<td>(PEPROCESS)AllocatePrMemory, PEPROCESS</td>
</tr>
<tr>
<td>ActiveProcess.DebugPort</td>
<td>PEPROCESS.DebugPort</td>
</tr>
<tr>
<td>ActiveProcess.UniqueProcessId</td>
<td>PEPROCESS.UniqueProcessId</td>
</tr>
</tbody>
</table>

6.1 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 \( \text{pts}(p) = \{a, c, b\} \) while the actual runtime targets are \( a \) and \( b \), then the algorithm is sound but not precise and thus there exist false positives. If \( \text{pts}(p) = \{a, c\} \) 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 some C programs from the SPEC2000 and SPEC2006 benchmark suites, and other open source C programs, to measure the soundness and precision percentage of KDD. Table 3 shows the characteristics of the used benchmarks in addition to their analysis result soundness and precision. We also show indications of memory, time and processor usage of running KDD on these benchmark programs. We used a commercial points-to analysis tool, CodeSurfer [32] (CodeSurfer is the only points-to analysis tool in the market that provides field- and context-sensitive analysis) to get estimation for the points-to sets of each variable in the used benchmark programs to compare results and performance to KDD. However, CodeSurfer results was not quite accurate, thus we manually verified each program to get an accurate estimation (retrieved points-to set). For programs that are less than 4 KLOC, we instrumented pointers manually. For larger programs we instrumented nearly 50% of the pointers. However, we could not measure precision for some programs because of their size. We also ran each program and monitored object allocation in the physical memory to get the actual runtime targets (relevant points-to set). Then, we used the below equation to calculate the precision. These results show that for the benchmark C programs analyzed by KDD, we achieve a high level of precision and 100% of soundness. This means that the resultant type graph accurately identifies the types of C program data structures even when void * pointers and pointer casting are used extensively. 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.

\[
\text{Precision} = \frac{\text{Relevant Points to Set} \times \text{Retrieved Points to Set}}{\text{Retrieved Points to Set}}
\]
6.2 Kernel Analysis

To illustrate the scale of the problem presented by C-based OSs, 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/object types), global variables and generic * used in their source code. Table 4 summarizes this 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.

Table 3. Soundness and Precision Results running KDD on a suite of benchmark C programs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>Pointer</th>
<th>Proc</th>
<th>Struct</th>
<th>AST T (sec)</th>
<th>AST M (MB)</th>
<th>AST C (%)</th>
<th>TG T (sec)</th>
<th>TG M (MB)</th>
<th>TG C (%)</th>
<th>P (%)</th>
<th>S (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>1272</td>
<td>286</td>
<td>43</td>
<td>19</td>
<td>22.7</td>
<td>21.5</td>
<td>19.9</td>
<td>73.3</td>
<td>12.3</td>
<td>17.6</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>equake</td>
<td>1515</td>
<td>485</td>
<td>40</td>
<td>15</td>
<td>27.5</td>
<td>25.4</td>
<td>20.4</td>
<td>87.5</td>
<td>14.2</td>
<td>21.1</td>
<td>98.6</td>
<td>100</td>
</tr>
<tr>
<td>gnu</td>
<td>2414</td>
<td>453</td>
<td>42</td>
<td>22</td>
<td>43.2</td>
<td>41</td>
<td>28.5</td>
<td>14</td>
<td>23</td>
<td>27</td>
<td>97.2</td>
<td>100</td>
</tr>
<tr>
<td>parser</td>
<td>11394</td>
<td>3872</td>
<td>356</td>
<td>145</td>
<td>305.2</td>
<td>191.2</td>
<td>76.7</td>
<td>661.4</td>
<td>107.8</td>
<td>74.3</td>
<td>94.5</td>
<td>100</td>
</tr>
<tr>
<td>vpr</td>
<td>17731</td>
<td>4592</td>
<td>228</td>
<td>398</td>
<td>316.1</td>
<td>298.7</td>
<td>80.2</td>
<td>1031.5</td>
<td>163.2</td>
<td>79</td>
<td>NA</td>
<td>100</td>
</tr>
<tr>
<td>gcc</td>
<td>222185</td>
<td>98384</td>
<td>1829</td>
<td>2806</td>
<td>3960.5</td>
<td>3756.5</td>
<td>93.5</td>
<td>12962</td>
<td>2200</td>
<td>94</td>
<td>NA</td>
<td>100</td>
</tr>
<tr>
<td>sendmail</td>
<td>113264</td>
<td>9424</td>
<td>1005</td>
<td>901</td>
<td>2017.2</td>
<td>1915.1</td>
<td>91.6</td>
<td>6609</td>
<td>10750</td>
<td>91.5</td>
<td>NA</td>
<td>100</td>
</tr>
<tr>
<td>bzip2</td>
<td>4650</td>
<td>759</td>
<td>90</td>
<td>14</td>
<td>82.3</td>
<td>78.1</td>
<td>45.5</td>
<td>271.6</td>
<td>44.2</td>
<td>42.9</td>
<td>95.9</td>
<td>100</td>
</tr>
</tbody>
</table>

KDD scales to the very large size of such OSs. KDD needed around 46 hours to analyze the WRK and around 72 hours to analyze the Linux kernel. Comparing KDD to KOP, KOP has to be run on a machine with 32 GB RAM. KDD has an improved performance of around 40% over KOP. The performance of KDD could be improved by increasing RAM and processing capabilities. We attempted to use CodeSurfer to analyze the WRK and Linux kernel to compare results and performance to KDD and KOP. However, CodeSurfer could not perform the analysis as it ran out of memory after several days of operation. 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. Re-generation of the graph is only necessary for different versions of a kernel where data structure changes may have occurred. 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. We manually compared around 74 generic pointers from WRK and 65 from the Linux kernel. Table 5 shows the results for few structures (space limits), showing 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 the 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 measured 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. By this we mean the generated type-graph is very close to the manually efforts.

6.3 Object-Graph for Security Monitoring

We modified our earlier-developed kernel security monitoring tool, CloudSec [6], to use the KDD-generated type-graph to traverse physical memory of running OSs, in order to construct a correct object-graph that identifies all the running instances of the data structures at a given memory snapshot. 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 Virtual Machines’ (VMs) memory from outside the VM itself, without installing any security code inside the VM using VMI. CloudSec uses memory traversal techniques to map the running objects based on manual profiles that describe the direct and indirect relations between structures. In this experiment, we used the generated type-graph to locate the dynamic objects by traversing the memory starting from the OS global variables and then followed 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 all of the running objects was around 12.5 minutes for a memory image of 43.5 MB on a 2.8 GHz CPU with 6GB RAM. To evaluate the mapping results, we considered the global variable PActiveProcessHead then followed pointer dereferencing until we covered different 43 object types (data structures) with their instances. We compared the results with the internal OS view using Windows Debugging tools [33], CloudSec successfully mapped and correctly identified the running kernel objects, with a low rate of false positives (around 1.5% in traversing balanced trees). This demonstrates that, for these 43 data structures monitored, our KDD-generated type-graph is in principle accurate enough for OS kernel data disambiguation to support security monitoring.

7. DISCUSSION

KDD is a static analysis tool that operates offline on 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 robust and accurate, and solves the null and void pointer problems with a high percentage of soundness and precision. Performing static analysis on the 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 e.g. we have the ability to systematically protect kernel data without the need to understand deep details about kernel data and behavior, as done to date. (ii) Performance Overhead; minimize the performance overhead in any further security applications, where a major part of the analysis process is done offline. If no static analysis were done, every pointers deference would have to be instrumented, which increase the performance overhead. (iii) Detecting Zero-Day Threats; we maximize the likelihood of detecting zero-day threats that target generic (via bad pointer dereferencing) or the 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 [25]. Brute force scanning tools identify dynamic running objects (data structure instances) at a given memory address using the data structure signature. (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. This is by inferring the target candidate type for each function pointer and thus decreasing 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 [9]. However in addition to the limitations discussed in the related work section: (i) the points-to sets of KOP are not highly precise compared to KDD, where they depend on the Heintze points-to analysis algorithm [16] which is used in compilers for fast aliasing. In addition, KOP computes transitive closures in order to perform the points-to analysis that increase the performance overhead of the analysis. 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 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.

To the best of our knowledge, our points-to analysis algorithm is the first points-to analysis technique that depends on the AST to provide interprocedural, context and field sensitive analysis. Buss et al. [23] has an initiative in performing points-to analysis based on the AST of the source code. However, their algorithm is field and context insensitive.

8. 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. In this paper, we presented KDD, 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. We implemented a proof-of-concept prototype of our tool. Our experiments with this prototype have shown that the generated type-graph is accurate and solves the generic pointer problem with high rate of soundness and precision. KDD processes large programs with acceptable CPU time and memory usage. 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-based operating system or similar program.

9. REFERENCES


<table>
<thead>
<tr>
<th>OS</th>
<th>Structure / GV (structure)</th>
<th>KDD</th>
</tr>
</thead>
</table>
| Linux    | thread_group (structure) | task_struct.thread_group[:task_struct.thread_group_leader.thread_group
thread_group.next:[list_head.next, task_struct.thread_group_leader.thread_group
thread_group.next:[list_head.next, task_struct.thread_group_leader.thread_group
| Windows  | ActiveProcessLinks (structure) | ActiveProcessLinks: [List_Entry, PsActiveProcessHead], ActiveProcessLinks.Flink: [List_Entry.Flink, PsActiveProcessHead.Flink], ActiveProcessLinks.Blink: [List_Entry.Blink, PsActiveProcessHead.Blink] |
| Windows  | LdtInformation (void*) | LdtInformation: [PVoid, PROCESS_LDT_INFORMATION] |
| Windows  | DirectoryTableBase (unsigned integer) | DirectoryTableBase: [MmCreateProcessAddressSpace:-1], DirectoryTableBase[0]: [PageDirectoryIndex, ULONG64], DirectoryTableBase[1]: [HyperSpaceIndex, ULONG64] |


M. Hind and A. Pioli, "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.


