Bridging the Semantic Gap Through Static Code Analysis

Christian Schneider Jonas Pfoh Claudia Eckert

Department of Computer Science
Technische Universität München
Munich, Germany
{schneidc,pfoh,eckertc}@in.tum.de

ABSTRACT

The semantic gap is a challenge inherent in all applications of virtual machine introspection (VMI). It describes the disconnect between the low-level state that the hypervisor has access to and its semantics within the guest. A common approach to bridge this gap is to utilize the debugging symbols of an inspected operating system kernel, although it is well understood that this information does not reflect the dynamic pointer manipulations that an operating system kernel performs at runtime.

In this work, we describe an analysis technique for capturing dynamic pointer manipulations and type casts in C code. Our approach analyzes the unmodified kernel source code to establish *used-as* relations between pointer types and to extract the arithmetic that is performed to transform a source pointer to a target address. We have implemented this technique in our VMI tool *InSight* for Linux to augment the type information retrieved from the debugging symbols. With this extended type information, our tool is able to cope with runtime pointer manipulations performed by the Linux kernel in a completely automated fashion and greatly eases the development of new VMI applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive Software*

General Terms

Security, Forensics

Keywords

Virtualization, Introspection, Digital Forensics, Intrusion Detection, Security

1. INTRODUCTION

When inspecting a virtual machine (VM) from the vantage point of the hypervisor, the entirety of the VM state is accessible. However, the hypervisor is missing the needed semantic information to properly interpret this state. This inherent lack of semantic information in the hypervisor is referred to as the *semantic gap* [4].

A VMI component that bridges the semantic gap by applying knowledge about the virtual hardware architecture or the guest operating system (OS) generates a useful "view" of the VM state that other VMI components can work with. It is therefore referred to as a *view generating component* (VGC) [11]. While building a VGC that uses the debugging information for a kernel is fairly straight forward, such a component is limited to finding statically typed objects. As a consequence, it will fail to handle dynamic pointer manipulations such as pointer arithmetic or type casts in a generic way. If the source code of the running kernel is available, which is the case for Linux, such dynamic behavior can be captured when performing a static code analysis.

In this paper, we describe a novel approach for identifying dynamic pointer manipulation in full C source code. We call this method a *used-as* analysis. In contrast to a points-to analysis [2, 8], it focuses on type usages instead of pointer locations. The used-as analysis reveals usages of structure or union fields and global variables in type contexts that contradict their declared types. In addition, it allows one to extract the pointer arithmetic that is performed to transform a pointer value to a target address.

We have implemented this analysis technique as an extension to *InSight*, a versatile VGC for Linux guests [15], to further improve its accuracy and coverage when reading kernel objects from memory. In applying the used-as analysis to the kernel sources, InSight establishes extended type relations and captures pointer arithmetic that the kernel would perform at runtime. This information augments the kernel debugging symbols InSight has been using so far and allows it to build an extended type graph of kernel data structures. With this extension, InSight is now able to automate the retrieval of objects referenced by incorrectly typed pointers, pointers stored as integer types, as well as generic pointers such as `void*`. In addition, InSight performs arithmetic operations on the pointer value to mimic kernel "pointer magic", for example, adding a type dependent offset or applying a bit mask with logical operations.

In summary, this paper makes the following contributions: First, we propose the used-as analysis for full C code in Section 3, which is essentially a type-centric modification of Andersen's points-to analysis. Second, we describe the implementation of the aforementioned analysis as an extension to our existing tool InSight in Section 4. To give evidence

```
1 struct list_head { struct list_head *next, *prev; };
2 struct foo { /* ... */ struct list_head list; };
3 struct list_head head_of_foo;
```

Figure 1: Organizing `struct foo` in a linked list by embedding a field of type `struct list_head`.

for the effectiveness of our methods, we describing some successful applications of our extended tool in Section 5. We complete this paper with an outlook in Section 6, a comparison to similar approaches in Section 7, and our concluding remarks in Section 8.

2. MOTIVATION

The goal of InSight is to find instances of kernel data structures in the (guest-)physical memory of an operating system. That is, it starts from a fixed set of kernel objects at well-known locations (i. e., global variables within the kernel space) and follows all pointer fields of these objects to further objects, and so on. In this way, the kernel objects resemble a directed graph where the objects represent the nodes and the pointer fields of all objects represent the edges.

The debugging symbols for the kernel contain information about the interconnection of kernel objects in form of pointer fields of structures to further data structures[1]. However, the C language in which OS kernels are mostly written allows to dynamically change pointer locations and types using pointer arithmetic and type casts. This is not reflected in the debugging symbols, rendering this information not only incomplete but sometimes even misleading. In the following, we give some examples for such "pointer magic" to illustrate the fundamental problem.

OS kernels organize objects in efficient data structures such as linked lists or hash tables, for example. A common implementation of these lists and tables works by defining generic "node" data structures along with auxiliary functions and macros to operate on them. These nodes are then embedded as a field of the data structure's definition to be organized in a list or table. To retrieve objects stored in such data structures, the auxiliary functions operate only on the embedded node but still make the embedding object available through address manipulations and type casts.

In the Linux kernel, for example, some data structure `foo` would embed a field of type `struct list_head` into its definition to arrange these objects in a doubly linked list.[2] An example of this code is given in Figure 1. This list is then accessed using the variable `head_of_foo` and operated on by the provided auxiliary functions and macros. They only operate on the `next` and `prev` pointers of field `list` which hold the addresses of field `list` *within* the next and previous objects, respectively. An illustration of this pointer linkage is shown in Figure 2. To get access to the object that embeds the `list` field, another macro is provided that subtracts the offset of field `list` from the pointer value and performs a cast to the requested type `struct foo*`.

This example involves several instances of dynamic pointer manipulations. First, the static type information for variable `head_of_foo` only indicates that the `next` and `prev` fields point to objects of type `list_head`, but there is no hint regarding the embedding `foo` objects. Second, once we have access to a `foo` object and follow the pointers of the embedded `list` field, we again loose awareness of the embedding objects of type `foo`. To make matters worse, many structures contain fields of type `list_head` as the heads to lists of objects with different types, so we cannot simply guess the correct type based on the embedding type. Third, the offsets that are subtracted from the `next` and `prev` pointer values are not always consistent. For example, the data structure `task_struct` arranges the process descriptors in a tree using two `list_head` fields `children` and `sibling`. Contradicting the common usage pattern, the pointer `children.next` does not point to the embedding field `children` of the first child's process descriptor but rather to field `sibling` of the first child. Analogous, `children.prev` points to the `sibling` field of the last child's process descriptor.

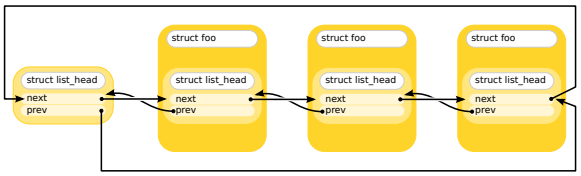

Figure 2: A doubly linked list utilizing `struct list_head` with three elements.

There are several other data structures that work in a similar way, such as singly linked lists or hash tables. Further examples of dynamic pointers include type casts from untyped pointers or integer values. In addition, some pointer or integer types might hold an address together with some flags or status bits stored in their least significant bits. These bits are masked out at runtime before the resulting address is dereferenced.

In order to reliably find *all* kernel objects in memory, we must be able to identify and reproduce this dynamic behavior. This can be achieved in two steps: First, a static source code analysis reveals all occurrences of pointer arithmetic and type casts. Second, this information is linked to the types and global variables found in the debugging symbols.

3. STATIC CODE ANALYSIS

In static code analysis, the *points-to* analysis, according to Andersen [2], tries to answer the question, "to which set of locations might a particular variable point to?" This method considers assignment statements and function invocations to keep track of the memory locations a pointer variable might hold. Since the pointers are tied to variables which only exist within a certain scope (which may also be the whole program in case of global variables), this type of analysis is rather data-flow driven and a points-to set is only valid within a pointer's scope. A variant of this method [8] has already been applied to identify kernel objects of the Windows kernel using an inter-procedural points-to analysis [3].

Recalling the motivation for our work from Section 2, the goal of our analysis is to identify dynamic pointer manipulations that are performed either for fields of data structures or for global variables. When we find an object of a certain type by following a pointer field from another object, that newly discovered object is not bound to any control-flow context or scope. For the purpose of our analysis, it resides

[1]With "structure" we refer to both `struct` and `union` types.
[2]For Windows, the corresponding structure is named `LIST_ENTRY` and works exactly in the same way as described for Linux's `list_head`.

in the global scope whether it was originally allocated on some heap, on some stack, or originated from a static data segment. So when we access a field of an object of a certain type, two questions arise: First, is this type's field *used as* a pointer to an object of a type that differs from its declaration? Second, how do we need to transform the value of this field in order to retrieve the object's address? It is exactly these questions that our used-as analysis will answer.

3.1 Used-As vs. Points-To Analysis

The approach of Carbone et al. [3] is based on the assumption that the actual data type stored in a pointer location is used at least once in some assignment statement within the inter-procedural control-flow of the pointer variable. As we have explained before, the objects read from memory do not have a particular context or scope. As a consequence, all possible program points using this pointer type must be considered when such a pointer is encountered. We exploit this observation as follows.

We use a method similar to a field-sensitive flow-insensitive points-to analysis. The key difference is that our analysis establishes used-as relations between structure or union fields as well as global variables on the one hand and data types on the other hand. That is, it identifies all types that a pointer or an integer type is used as. In addition, it records all static pointer arithmetic that is applied in such situations to retrieve the target address from a source pointer. These related types together with their pointer arithmetic represent the candidate types that have to be considered when following the field of an object or reading a global variable from memory. Such an approach contrasts the conservative pointer treatment of a static type system, for example, as performed by a compiler. However, completeness of the view is crucial during introspection. Consequently we see it justified to over-approximate and consider all possible interpretations of a pointer at the cost of possible ambiguities.

Our approach has two advantages over an inter-procedural points-to analysis. First, it only requires an intra- instead of an inter-procedural analysis, leading to a reduced complexity. This is due to the fact that our analysis needs to find a type usage only once per type, not once per pointer variable. Second, it detects type usages not only in assignment statements, but in all other expressions that might change a pointer's type, such as type casts or return statements. Consider the following code fragment:

```
struct A { /* ... */ };
struct B { void *data; };

struct A *pa = malloc(sizeof(struct A));
void *p = malloc(sizeof(struct B));
((struct B*)p)->data = pa;
```

Most points-to analyses would fail to detect that the field `data` of `struct B` is in fact used as a pointer to `struct A`, because the inline type cast does not change the location of `p`, and `p` does not explicitly point to a `struct A` object. In contrast, our used-as analysis correctly recognizes that the type cast followed by a pointer dereference through the arrow operator '`->`' constitutes a runtime type usage.

Another more technical difference is that many points-to analysis systems transform the source code into simplified statements or work on some lower-level internal representation of the code. Our analysis works on full (pre-processed) C code[3] and does not perform any preliminary transformation steps.

3.2 Step 1: Points-To Analysis

Since our analysis is type centric, the used-as relations are only relevant for two types of symbols: Global variables of pointer or integer types and structure fields of these types. However, such symbols are often assigned to local variables before the actually interesting type usage occurs. Thus, we first perform a field-sensitive, intra-procedural, and control-flow insensitive points-to analysis of the source code.

We initialize the points-to map by examining all assignment expressions and mapping the left-hand side (lvalue[4]) to the right-hand side. In addition, we label each such mapping with the dereference level of the lvalue (i. e., the number of $*$ operators). Let i be the dereference level, x be the lvalue and $e(y)$ be a legal expression in C that involves variable y, then we denote such a mapping with the relation $x \rightarrow^i \{e(y)\}$. Consider the following example:

```
void *g;

void func() {
    struct A { void *p; } a, *pa;
    void *x, **y;
    x = g;
    y = &x;
    *y = a.p;
    pa = (struct A*) x;
}
```

For this code, the initialization yields $x \rightarrow^0 \{g\}$, $y \rightarrow^0 \{\&x\}$, $y \rightarrow^1 \{a.p\}$, $pa \rightarrow^0 \{(\texttt{struct A}*)x\}$.

Next, we derive the transitive closure of the map as follows. For all structure fields $s.f$ and global variables y that appear in expressions $e(s.f)$ or $e(y)$ some variable x points to, e. g., $x \rightarrow^i \{e_1(y)\}$, we find all mappings of the form $z \rightarrow^j \{e_2(*^i x)\}$. Here $*^i x$ denotes i occurrences of the star operator to x. For each mapping found, we add the expression $e_2(e_1(y))$ to the expressions being pointed to by z, resulting in $z \rightarrow^j \{e_2(*^i x), e_2(e_1(y))\}$. Note that we require that $y \neq z$, in other words, we disallow recursive expressions. For this reason, we also disregard all expressions with arithmetic assignment operators, such as '`+=`'.

In order to correctly handle indirect assignments of variables as in '`x = &y; *y = a.p;`', we proceed as follows. For mappings $x \rightarrow^i \{e_1(y)\}$ with $i > 0$, we also look for mappings of the form $x \rightarrow^j \{\& e_2(z)\}$ with $j = i - 1$. If the expression $e_2(z)$ yields a valid lvalue, we add $e_2(z) \rightarrow^0 \{e_1(y)\}$ to our map and treat it in the same way as any other points-to mapping. We repeat these steps for all mappings that were added in the previous round until no more mappings are added and the transitive closure is complete. For the previous code fragment, the resulting map would be:
$x \rightarrow^0 \{g, a.p\}$, $y \rightarrow^0 \{\&x\}$, $y \rightarrow^1 \{a.p\}$,
$pa \rightarrow^0 \{(\texttt{struct A}*)x, (\texttt{struct A}*)a.p, (\texttt{struct A}*)g\}$

3.3 Step 2: Establishing Used-As Relations

To establish the used-as relations, we compare the type each global variable or structure field is used as to its declared type. For this comparison, we also take the points-to map into account that has been generated in the previous step to detect indirect type usages by means of local variables. A type usage may occur in the context of assignment

[3] Including many GCC extensions.
[4] A value that has an address and can be assigned to.

```
1  struct A { int value; struct A *next; };
2  struct B { void *data; }
3
4  struct A* func1(struct A *a) { return a; }
5
6  struct A* func2() {
7      struct B b;
8      struct A a = { 0, b.data };     // struct initializer
9      struct A *pa = b.data;          // pointer initializer
10     pa = b.data;                    // assignment
11     a = *((struct A*)b.data);       // dereference (*)
12     ((struct A*)b.data)->value++;   // dereference (->)
13     pa = func1(b.data);             // function parameter
14     return b.data;                  // return statement
15 }
```

Figure 3: Various usage patterns that establish a used-as relation between field `data` of `struct B` and `struct A*`.

statements, initializers, pointer dereferences after type casts, function parameters, and return statements. An example for each of these usages is given in Figure 3. For all of the cases shown in the example, our used-as analysis would come to the same conclusion: Field `data` of `struct B` having type `void*` is in fact used as a pointer of type `struct A` with no additional pointer arithmetic. This relation is stored in the type information for structure `B` and will be considered whenever the field `data` of such an object is accessed.

Special care needs to be taken for structures that are embedded into other structures. Consider the example in Figure 1 once again. The field `list` in line 2 is defined as an embedded `struct list_head` within structure `foo`. If we were to propagate the used-as relations of `list.next` and `list.prev` to all other fields of type `struct list_head` in all other structures, we would essentially mix up all types that are arranged in doubly linked lists, leading to impractically high numbers of candidate types to follow. Therefore, whenever one structure embeds another, we create a unique copy of the embedded structure type for the embedding type. We then assign any used-as relations for its fields to these copies. In the same way we create copies for structure or union types of global variables. This results in variable and type context-sensitive used-as relations which vastly improves the type accuracy for locating kernel objects.

4. IMPLEMENTATION

We have implemented the proposed used-as analysis as an extension to our existing tool, InSight [15]. For a Linux kernel, InSight is able to read kernel objects from physical memory and to follow the fields of structure objects, dereference pointers and access array elements. It exposes these objects through a convenient JavaScript interface to arbitrary VMI applications, such as intrusion detection, forensic analysis, malware analysis, or kernel debugging.

InSight uses the debugging symbols of the inspected kernel to locate global variables in the kernel's address space and derive the layout of data structures. The extended type information gathered from the used-as analysis now augments the static type information, allowing InSight to consider all possible pointer usages of any field or variable that occurred anywhere in the kernel's source code.

4.1 Consolidating Type Information

The foundational knowledge about the types and global instances of kernel objects InSight uses is contained in the kernel's debugging symbols. They completely reflect all applied compiler directives and linker flags that have influence on the location and the alignment of types and variables. For a Linux guest, the debugging symbols can be obtained by re-compiling the kernel with specific compiler flags for symbol generation, which neither influences the performance nor the semantics of the resulting kernel image.

The compiler generates the debugging symbols per translation unit. As a consequence, they contain a high degree of redundant type information, because any type that is used in n translation units appears n times in the debugging symbols. This is undesirable for our purpose as the redundant types require an excess of memory, but more importantly, they counter our effort of linking the used-as relations to a particular type. We solve this problem as follows.

When InSight processes the debugging symbols for the first time, it creates hash values for all data types based on their name, size, signedness (for integer types), fields (for structures or unions), and parameters (for function pointers). Equivalent types are recognized based on these hashes and are merged into single instances. For a recent 3.0 Linux kernel, this reduces the number of unique types to less than three percent (about 61k) of their original count of over 2.2 million. The consolidated types are then saved in a custom format to minimize the effort for future use.

4.2 Source Code Level Analysis

The consolidated type information extracted from the debugging symbols builds the initial knowledge basis for InSight. To capture the dynamic pointer treatments of the kernel, InSight parses the kernel's source code in a second step, performs the used-as analysis, and builds an extended type graph from the collected information to reflect such dynamic treatments. Through this combination of debugging symbols and static code analysis, InSight achieves a high object coverage and type accuracy.

The Linux kernel comes with a highly sophisticated build system based on the GNU `make` utility. Depending on the kernel configuration and system environment, the build system selects proper compiler flags, macro definitions, and include paths to compile the source files. In order to avoid having to mimic this complex system, we instead use a small wrapper script for the GNU C compiler that stores the preprocessed source files during compilation in a separate directory. Thus, we re-compile the guest kernel with debugging flags enabled and our wrapper script set as the compiler. This assures that the code being compiled exactly corresponds to the code that InSight analyzes later on.

In our experiments, the used-as analysis of over 20 million lines (584 MB) of pre-processed C code took less than 20 minutes on a standard PC[5].

4.3 Disambiguating Candidate Types

InSight uses a simple strategy to apply the used-as type relations to kernel objects: If a pointer has exactly one candidate type, this candidate overrides the type defined in the debugging symbols. In case the analysis has revealed several candidate types (typically for less than 0.5% of all types), the defined type is used as a fail-safe default; it is not overridden. However, the user can always query the available candidates and choose the one that he sees best fit.

[5]Intel Core 2 Quad Q9550 (2.83 GHz), 8GB RAM

```
1  function lsmod() {
2      // request instance of the list's head
3      var head = new Instance("modules");
4      // iterate over all modules
5      var m = head.next;
6      while (m.MemberAddress("list") != head.Address()) {
7          print(m.name + " " + m.args);
8          m = m.list.next;
9      }
10 }
```

Figure 4: JavaScript code to print the list of loaded modules for a Linux guest.

The scripting engine can also be used to try all candidate types for a pointer and perform sanity-checks on the returned objects in an automated fashion. For example, a "sane" object should be located in the kernel's address space, its address should be aligned to a four byte boundary, pointers to further objects should follow the same rules, and so on. By organizing such functionality in script libraries, it can easily be shared among multiple VMI applications to reduce the effort for the individual approaches.

5. APPLICATION

InSight strives to interpret the entire kernel memory of a running guest OS in the same way the guest kernel does. It is very flexible in that it provides access to the VM state through a powerful shell interface as well as through a scripting engine that allows interaction with kernel objects as JavaScript objects. This completely decouples the VMI application from the view generation process, making InSight a universal tool for various applications such as intrusion detection and forensic analysis, as well as for kernel debugging.

The scripting engine allows users to write JavaScript code for interacting with kernel objects and performing complex tasks. The user may access global variables by requesting an `Instance` object of a variable by using its name. If the resulting object represents a structure, its fields can be accessed by their name using the "dot" notation just like any other property of a JavaScript object. Per default, all fields having pointer types are automatically dereferenced. In addition, an `Instance` object provides functions to access the meta data of the represented kernel object, change its type, manipulate its address, and inspect the candidate types from the extended type graph.

Figure 4 shows a simple example script that prints all loaded kernel modules for a Linux guest. Note that the `Instance` object that is retrieved from the global variable `modules` in line 3 actually is of the type `list_head`. This scenario corresponds to our example in Figure 1 where a variable of type `list_head` represents the head of a list. Through the used-as analysis, InSight has determined that the `next` field of this variable in fact points to `struct module` objects when a certain offset is subtracted. The same is true for the field `list` of the instances of `struct module` that are returned by following `m.list.next` in line 8. This illustrates how InSight is effectively able to hide the "pointer magic" of the doubly linked lists from the user by providing a very simple and intuitive interface.

InSight allows one to perform various types of analysis and is deployed in several projects for VMI-based intrusion detection within our research group. We describe some of these applications in the following.

5.1 Periodic Analysis

One way of monitoring a system for intrusions is to perform an analysis of the guest in regular intervals and report any suspicious findings. If the interval is chosen to be sufficiently long and the analysis can be performed quickly, this method typically incurs only a small overhead.

An approach for VMI-based intrusion detection that has been described by various authors [7, 9, 10] is called *lie detection*. It works by generating two different views simultaneously, one inside of the guest OS and one with the help of VMI techniques. These views are then compared with each other and any discrepancy between them is taken as an indicator for malicious activities. As a proof of concept, we have implemented a lie detector for running processes and loaded kernel modules on a live guest and successfully revealed rootkits that are trying to hide their presence.

In addition, we experimented with new ways of detecting intrusions on the kernel level. For this purpose we created a series of sequential memory dumps of the guest while inducing normal and malicious activities. Here InSight was used to perform off-line analysis of the collected states by to gathering information about running processes, finding the location of the code sections of loaded kernel modules to detecting kernel code patching, and revealing changes to the system call table, among other things. This approach showed that InSight is also very much capable of performing forensic analysis.

5.2 Event-driven Analysis

An alternative method for system analysis monitors and reacts to system events. In such cases, a VMI-based approach manipulates the system in such a way that the events of interest cause a trap to the hypervisor [12]. Before the hypervisor returns control to the guest, the VGC collects the required data from the current state of the guest. Depending on the frequency of such events, it is crucial to avoid any unnecessary overhead for the analysis to minimize overall performance degradation.

We have combined InSight with a VMI-based framework for system call tracing, called Nitro [13]. Nitro uses InSight to augment the low-level parameters of system calls it monitors to produce a much richer output compared to plain system call numbers and pointers. For example, InSight is able to determine the concrete type of a process's file handle in calls to the "read" and "write" system calls and output corresponding information, for example, the IP addresses of a connected socket or the file name of a regular file. Other system calls are augmented similarly. This project has shown that the separation of InSight into a back-end and a front-end helps to minimize the per-analysis overhead. The parameter augmentation contributes only to a small degree to the total overhead of the complete framework.

6. LIMITATIONS AND FUTURE WORK

While the used-as analysis has vastly improved the usefulness of InSight, the over-approximation of candidate types for pointers now possibly leaves the user with a small percentage of types having multiple candidates to choose from. InSight does not yet offer any strategy to disambiguate these candidates. Our goal for the future is to provide a mechanism that tests all candidates and automatically chooses the most likely type as follows.

When analyzing the kernel source code, we can not only extract the pointer type usage but also keep track of the usage of other (non-pointer) fields of data structures. This information could then be used to infer invariants for individual fields. For example, if an integer field is only set to values of a specific enumeration type, an invariant for the structure would be that this particular field must always hold one of these values. Such invariants would allow to weight the "soundness" of multiple candidate objects and to decide for one of them.

7. RELATED WORK

As bridging the semantic gap is a requirement for all VMI components, this challenge has been addressed in several specific ways. Many approaches rely on the debugging symbols, manually supplemented with expert knowledge where this is required (e. g., [7, 6, 14]).

The only other approach we are aware of that applies a source code level analysis for view generation is the KOP system developed by Carbone et al. [3]. It follows a similar approach to InSight and performs a static points-to analysis of Windows Vista's source code to identify kernel objects in memory. KOP works on a simplified representation of the code to construct an extend type graph and applies heuristics to handle ambiguous or incomplete types. As described in Section 3.1, the points-to analysis might not always detect all type usages. Another limitation of this system is that the only supported form of pointer arithmetic is the addition of a fixed offset. Even though the authors do not detail the user and programming interface of KOP, we see a clear benefit in the flexible interfaces that InSight provides.

Another very interesting approach to bridging the semantic gap was introduced by Dolan-Gavitt et al. in the form of Virtuoso [5]. While InSight automates certain complex tasks necessary for complete view generation, Virtuoso automates the generation of an entire VGC by "training" a component to retrieve the desired state. This is a very interesting approach as building a VGC is a complex and time consuming task as we have experienced. However, automating the generation of a VGC that is capable of generating a view of the guest kernel that is as complete as the one created by InSight is well beyond Virtuoso's scope.

8. CONCLUSION

We have presented an analysis technique to capture dynamic pointer manipulations in full C source code. In contrast to the rather data-flow oriented points-to analysis, our analysis is type centric and establishes used-as relations between global variables, structure fields and pointer types. In order to support the complete emulation of dynamic pointer manipulations, our approach also extracts any pointer arithmetic that is performed to yield the target address from a pointer value.

The implementation of this kind of analysis in our tool InSight augments the type information it was already using before. This extensions now enables InSight to consider all feasible interpretations of a field or variable, leading to a vastly improved completeness and accuracy when performing introspection tasks. Finally, we offered evidence of the effectiveness of our method by outlining several successful applications of InSight within our research group.

InSight is available as an open source tool [1] to enable the fast and intuitive development of new VMI and forensic applications.

9. REFERENCES

[1] InSight project website. `https://code.google.com/p/insight-vmi/`.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proc. of 16th Conf. on Computer and Communication Security*, CCS '09, pages 555–565. ACM, 2009.

[4] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, page 133. IEEE, 2001.

[5] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. of the IEEE Symp. on Security and Privacy*, May 2011.

[6] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proc. of Int. Conf. on Automated Software Engineering*, ASE '10, pages 417–426, New York, NY, USA, 2010. ACM.

[7] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. of NDSS*, pages 191–206, 2003.

[8] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proc. of conf. on Programming language design and implementation*, PLDI '01, pages 254–263. ACM, 2001.

[9] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proc. of the 17th conf. on Security symp.*, pages 243–258, Berkeley, CA, USA, 2008. USENIX.

[10] L. Martignoni, A. Fattori, R. Paleari, and L. Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Proc. of 13th Int. Conf. on Recent Advances in Intrusion Detection*, RAID'10, pages 297–316. Springer, 2010.

[11] J. Pfoh, C. Schneider, and C. Eckert. A formal model for virtual machine introspection. In *Proc. of 2nd Workshop on VM Sec.* ACM, 2009.

[12] J. Pfoh, C. Schneider, and C. Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Proc. of the 4th Int. Conf. on Emerging Security Information, Systems and Technologies*, Venice, Italy, July 2010. IEEE.

[13] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, LNCS. Springer, Nov. 2011.

[14] J. Rhee and D. Xu. LiveDM: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Technical report, Purdue University, 2010.

[15] C. Schneider, J. Pfoh, and C. Eckert. A universal semantic bridge for virtual machine introspection. In *Information Systems Security*, volume 7093 of *LNCS*, pages 370–373. Springer, 2011.