DDE: Dynamic Data Structure Excavation

Asia Slowinska
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
asia@few.vu.nl

Traian Stancescu
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
tsu500@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
herbertb@few.vu.nl

ABSTRACT

Dynamic Datastructure Excavation (DDE) is a new approach to extract datastructures from C binaries without any need for debugging symbols. Unlike most existing tools, DDE uses dynamic analysis (on a QEMU-based emulator) and detects data structures by tracking how a program uses memory. Its results are much more accurate than those of previous methods.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Algorithms, Design, Experimentation

Keywords

binary, detecting data structures, dynamic analysis

1. INTRODUCTION

Debugging and reverse engineering of C binaries is difficult, especially in the absence of debugging symbols. Since programs tend to be developed around the data structures, they arguably represent the most important information that we need to recover. Unfortunately, data structure recovery is exceedingly hard. Even the most state of the art disassemblers and decompilers (like IDA Pro [9], CodeSurfer [1] and boomerang [10]), while fairly good at recovering code blocks, are hopeless at identifying data structures.

Data structure recovery is difficult, due to the gap between how data appears in the source and in the binary. The compilation process turns all variables into chunks of anonymous bytes. Data structure excavation is the art of mapping them back into meaningful data structures. To our knowledge, no existing work can do this.

In this paper, we sketch our solution for data structure excavation in x86 C binaries. Unlike most other approaches, we build DDE primarily on dynamic rather than static analysis following the simple intuition that memory access patterns reveal much about the layout of the data structures. DDE is able to recover most data structures in arbitrary (gcc-generated) binaries with a very high degree of precision. While it is too early to claim that the problem of data structure identification is solved, our work advances the state of the art significantly. For instance, we are the first to extract:

- precise data structures on both heap and stack;
- not just aggregate structures, also individual fields;
- complicated structures like nested arrays.

Due to space limitations, we describe our techniques at a fairly high level. Readers interested in all details are referred to our technical reports [14, 13]. All dynamic analysis techniques were implemented in an instrumented processor emulator based on Qemu [4]. Since the processor emulator is available only for Linux, the implementation is also for Linux. However, the approach is not specific to a particular operating system.

Data structures detected by DDE can be used to partially generate symbol tables on the fly [13], which in turn aids debugging, forensics and reverse engineering. In addition, DDE could allow us to retrofit security onto existing binaries. Specifically, the information gathered by the analysis can be used to protect legacy binaries against buffer overflows [13].

2. ARCHITECTURE

DDE recovers data structures by observing how memory is *used* at runtime. In the CPU, all memory accesses occur via pointers either using direct addressing or indirectly, via registers. The intuition behind our approach is that memory access patterns provide clues about the layout of data in memory. For instance, if `A` is a pointer, then a dereference of `*(A+4)` suggests that the programmer (and compiler) created a field of size 4 at `A`. Intuitively, if `A` is a function frame pointer, `*(A+8)` and `*(A-8)` are likely to point to a function argument passed via the stack, and a local variable, respectively. Likewise, if `A` is an address of a structure, `*(A+4)` presumably accesses a field in this structure, and finally, in the case of an `int[]` array, `*(A+4)` is its second element. Distinguishing between these three scenarios is one of the challenges we need to address. In this section, we discuss the major problems, and in Section 2.1 we explain how we tackle them.

Memory allocation context. Our work aims to analyse a program's use of memory, which includes local function variables allocated on the stack, memory allocated on heap, and static variables. Both the runtime stack and heap are reused constantly, and so a description of data structures here needs to be coupled with a *context*.

APSys 2010, August 30, 2010, New Delhi, India.

```
typedef struct {
 int x;
 int y;
} elem_t;

void fun() {
 elem_t elem, *pelem;
 elem.x = 1;
 elem.y = 2;
 pelem = &elem;
 pelem->y = 3;
}
```

```
 <fun>:
[1] push %ebp
[2] mov %esp, %ebp
[3] sub $0x10, %esp
[4] mov $0x1, -0xc(%ebp)
[5] mov $0x2, -0x8(%ebp)
[6] mov -0x4(%ebp), %eax
[7] mov $0x3, 0x4(%eax)
[8] leave
[9] ret
```

Figure 1: The function initializes its local variable `elem`. Pointer `pelem` is located at offset `-4` in the function frame, and structure `elem` at `-0xc`. Instructions 4 and 5 initialize `x` and `y`, respectively. Register `eax` is loaded with the address of `pelem` in instruction 6, and used to update field `y` in 7.

For the stack, each invocation of a function usually holds the same set of local variables and therefore start addresses of functions are sufficient to identify function frames. A possible exception occurs with memory allocated by calls to functions like `alloca`, which *may* depend on the control flow. As a result, the frames of different invocations could differ. DDE handles these cases in a generic way, but we refer to [14] for details.

Heap memory, however, is more complicated. Consider a `my_malloc` wrapper function which invokes `malloc` and checks whether the return value is null. Since `my_malloc` can be used to allocate memory for various structures and arrays, we should not associate the memory layout of a data structure allocated by `my_malloc` to `my_malloc` itself, but rather to its caller. As we do not know the number of such `malloc` wrappers in advance, we associate heap memory with a call stack (typically, the top 3 or 4 function calls are sufficient for accurate identification).

Static memory, finally, is not reused, and so can be uniquely identified solely with its address.

Pointer identification. To analyze memory access patterns, we need to identify pointers in the running binary. Moreover, for a given address `B=A+4`, we need to know `A`, the *base* pointer from which `B` was derived. However, on architectures like `x86`, there is little distinction between registers used as addresses and scalars. Worse, the instructions to manipulate them are the same. We only know that a particular register holds a valid address, when it is dereferenced. In our instrumented emulator, we therefore explicitly track how new pointers are derived from existing ones.

Missing base pointers. Recall that we recover data structures by observing memory accesses: new structure fields are detected when they are referenced from the structure base. However, structure fields are sometimes used without reference to a pointer to the data structure, and thus we may overlook the link between the fields. Figure 1 illustrates the problem. Notice that field `elem.y` is initialized via the frame pointer register `ebp` rather than the address of `elem`. Only the update instruction `7` hints at the existence of the structure. Otherwise, we would characterize this memory region as composed of 3 separate variables: `pelem`, `x`, `y` (on the other hand, since in that case the program does not actually use the connection between the fields `x` and `y`, this partially inaccurate result would be innocuous).

Multiple base pointers. Another issue is that memory locations can be accessed through multiple base pointers, so we need to decide on the most appropriate one. Observe that field `elem.y` from Figure 1 is already referred to using two different base pointers, the frame pointer `ebp` and `pelem` (`eax`). Even though this particular case seems tractable (as `pelem` is itself based on `ebp`), the problem in general is knotty. For instance, programs often use functions like `memset` and `memcpy` to initialize and copy data structures. Such functions access all bytes in a structure sequentially, typically with a stride of one word. Clearly, we should not classify each access as a separate word-sized field. In fact, this is a serious problem for all even the most advanced approaches to date (e.g., DIVINE [2]). In our opinion, treating such functions in a special way (by blacklisting, say) is a bad solution, as it will handle the known functions, but not similar ones that are part of the application itself. Instead, as we shall see DDE uses a heuristic that lets us dynamically select the "less common" layout. In other words, it favours data structures with different fields over an array of integers.

Code coverage. As our analysis is performed dynamically, the accuracy increases if we execute more of the program's code paths. Code coverage techniques (using symbolic execution and constraint solving) force a program to execute all/most of its code [6]. We do not discuss this further in this paper. Recent work at EPFL explains how to do this for binaries [7].

2.1 Our approach

2.1.1 Function call stack

As a first step in the analysis, our technique keeps track of the function call stack. As DDE runs the program in an instrumented processor emulator, it can dynamically observe `call` and `ret` instructions, and the current position of the runtime stack. A complicating factor is that sometimes `call` is used not to invoke a real function, but rather only to push the return address. Similarly, not every `ret` has a corresponding `call` instruction.

We define a *function* as the target of a `call` instruction which returns with a `ret` instruction. Values of the stack pointer at the time of the call and at the time of the return match, giving a simple criterion for detecting uncoupled `call` and `ret` instructions. Note that a function reached by means of a `jump` instruction is merged with the caller. We discuss impact of that on the analysis in [14].

2.1.2 Pointer tracking

We determine base pointers dynamically by tracking the way in which new pointers are derived from existing ones, and observing how the program dereferences them. In addition, we extract *root* pointers that are not derived from any other pointers. Root pointers initialize statically allocated memory, heap and stack.

For the reason of pointer tracking, we extended the processor emulator so that each memory location has a *tag*, `bp_memtag(addr)`, which stores its base pointer. In other words, a tag specifies how the address of a memory location was calculated. Likewise, if a general purpose register holds an address, an associated tag, `bp_regtag(reg)`, identifies its base pointer.

We first present tag propagation rules, and only afterwards explain how root pointers are determined.

When a new root pointer `A` is encountered, we set `bp_memtag(A)` to a constant value `root` to mark that `A` has been accessed, but not derived from any other pointers. When a pointer `A` (root or not) is loaded from memory to a register `reg`, we set `bp_regtag(reg)` to `A`.

The program may now manipulate the pointer using pointer arithmetic (e.g., `add`, `sub`, or `and`). To simplify the explanation, we as-

sume the common case, where pointers are manipulated completely *before* they are stored to memory, i.e., the intermediate results of pointer arithmetic operations are kept in registers only. This is not a limitation; it is easy to handle the case where a program stores the pointer to memory first, and then manipulates and uses it later.

During pointer arithmetic, we do *not* update the `bp_regtag(reg)`, but we do propagate the tag to destination registers. As an example, let us assume that after a number of arithmetic operations, the new value of `reg` is `B`. Only when the program dereferences `reg` or stores it to memory, do we associate `B` with its base pointer which is still kept in `bp_regtag(reg)`. In other words, we set `bp_memtag(B)` to `A`. This way we ensure that base pointers always indicate valid application pointers, and not intermediate results of pointer arithmetic operations.

Extracting root pointers. We distinguish between three types of root pointers: (a) those that point to statically allocated memory, (b) those that point to newly allocated dynamic memory, (c) the start of a function frame which serves as a pseudo root pointer for the local variables.

Dynamically allocated memory. To allocate memory at runtime, user code in Linux invokes either one of the memory allocation system calls (e.g., `mmap`, `mmap2`), or it uses one of the `libc` memory allocation routines (e.g., `malloc`). Since each memory region is analyzed as a single entity, we need to retrieve their base addresses and sizes. DDE uses the emulator to intercept both. Intercepting the system calls is easy - we need only inspect the number of each syscall made. For `libc` routines, we determine the offsets of the relevant functions in the library, and interpose on the corresponding instructions once the library is loaded.

Statically allocated memory. Root pointers to statically allocated memory appear in two parts of an object file: the data section which contains all variables initialized by the user - including pointers to statically allocated memory, and the code section - which contains instructions used to access these data. To extract root pointers, we initially load pointers stored in well-defined places in a binary, e.g., `ELF` headers, or relocation tables, if present. Next, during execution, if an address `A` is dereferenced, `bp_memtag(A)` is not set, and `A` does not belong to the stack, we conclude that we have just encountered a new root pointer to statically allocated memory. Later, if we come across a better base pointer for `A` than `A` itself, `bp_memtag(A)` gets adjusted.

Stack memory. Function frames contain arguments, local variables, and possibly temporary data used in calculations. Typically, local variables are accessed via the function frame pointer, `EBP`, while the remaining regions are relative to the current stack position (`ESP`).

As we do not analyze temporary variables, we need to keep track of pointers rooted (directly or indirectly) at the beginning of a function frame only (often, but not always, indicated by EBP). Usually, when a new function is called, 8 bytes of the stack are used for the return address and the caller's `EBP`, so the callee's frame starts at `(ESP-8)`. However, other calling conventions are also possible [14]. This means that we cannot determine where the function frame will start. To deal with this uncertainty, we overestimate the set of possible new base pointers, and mark all of them as possible roots. Thus, we emphasise that DDE does not rely on the actual usage of the `EBP` register. If, due to optimizations, `EBP` does not point to the beginning of the function frame, nothing bad happens.

2.1.3 Multiple base pointers

As a memory location `A` is often accessed through multiple base pointers, we need to pick the most appropriate one. Intuitively, selecting the base pointer that is *closest* to the location, usually increases the *number of hops* to the root pointer, and so provides a more detailed description of a (nested) data structure.

However, functions like `memset` and `memcpy` often process composite data structures. These functions are completely unaware of the actual structure and simply access the memory in word-size strides. Thus, for 32 bit machines, such functions continuously calculate the next address to dereference by adding 4 to the previous one covering the entire data structure in 4 byte strides. By applying the aforementioned heuristic of choosing the closest base pointer, we could easily build a meaningless recursively nested data structure.

For `structs` the solution is often simple. When the program accesses the memory twice, once with constant stride equal to the word size (e.g., in `memset`) and once in a different manner (when the program accesses the individual fields), we should pick the latter. In arrays, however, multiple loops may access the array. To deal with this problem, we use a similar intuition and detect arrays and structures dynamically with a heuristic preference for non-regular accesses and/or accesses at strides not equal to the word size. For instance, if a program accesses a chunk of memory in two loops with strides 4, and 12, respectively, we will pick as base pointers those addresses that correspond to the latter loop. Intuitively, a stride of 12 is more likely to be specific to a data structure layout than the generic 4. Our current array detection introduces three kinds of loop accesses: (1) accesses with non-constant stride, e.g., an array of strings, (2) accesses with a constant stride not equal to the word-size, e.g., 1 or 12, and (3) accesses with stride equal to the word-size. Our heuristic, then, is as follows. First select the base pointers in the best possible category (lower is better), and next, if needed, pick the base pointer closest to the memory location.

2.1.4 Array detection

As indicated above, DDE needs to detect memory accesses in loops. Specifically, we should detect how loops access arrays. This is quite hard and a general solution must handle the following difficult cases: (a) multiple loops placed in sequence, (b) nested sequences of loops, (c) inner loops and outer loops not iterating over the same array, and (d) first and last array element handling *outside* the loop.

In real code, there are two popular schemes of deriving array element addresses: (1) relative to the base of an array, realized in instructions like `elem=array[i]`, and (2) relative to the previous element, `new=*(pprev++)`. Our algorithms handle both cases. However, for simplicity's sake, we discuss the latter only, when the array is accessed in the most inner loop and leave the discussion of the alternative case to [14].

`New=*(pprev++)` access. DDE identifies each loop with an id (`lid`) which it assigns to the loop head at runtime when the back edge is taken for the first time. At this point this loop head is pushed on a stack. So, if a loop executes just once and never branches back for a second iteration, it does not get a new `lid`. The current loop head is pushed on a stack and DDE assigns the top `lid` as a tag to each byte of memory the code accesses. Thus memory accesses in the first iteration of a loop get the parent `lid`. Tags are kept similarly to `bp_memtag`, in the emulator.

DDE's core algorithm for detecting arrays is as follows: when a pointer `B`, derived from base pointer `A`, is dereferenced in iteration i, while `A` was dereferenced in a previous iteration, DDE treats `A` as a likely array element. It stores information about the array in

the loop head. The more iterations are executed, the more array elements DDE discovers.

Because we often access the first and last elements in an array outside the loop, DDE explicitly checks for extending the array. First, it looks for earlier memory accesses at the base pointers used to recursively derive the first element of the array, and checks whether they have a `lid` that is just below the top of the stack. Second, it looks for memory location based at the last element of the array, and checks whether it has the same `lid`.

In addition, DDE should also find arrays accessed in a non-sequential way. If an instruction in the loop accesses a set of addresses, we check whether they can be mapped to a linear space, `stride * x + offset`. This is similar to the array detection in Polyglot [5]. However, this basic approach does not work well if the loop body has multiple if/switch branches, and so multiple instructions accessing the same array. To deal with this issue, we check whether array intervals associated with instructions are interleaving, and if necessary take the sum of them.

Summarizing, we use the combination of the two aforementioned methods: detecting sequential, and non-sequential array access. This enables DDE to detect nested arrays, hash tables and many other complex cases. (Again, see [13] for details.) The array detection techniques in DDE are more complicated than what we sketched. While in rare cases DDE may overestimate the size of the array, its heuristics appear to work well in practice.

2.1.5 Final mapping

Having detected arrays and the most appropriate base pointers, DDE finally maps the analyzed memory into meaningful data structures. For a memory chunk, the mapping starts at a root pointer and reaches up to the most distant memory location still based (directly or indirectly) at this root. For static memory, the mapping is performed at the end of the program execution. Memory allocated with `malloc` is mapped when it is released using `free`, while local variables and function arguments on the stack are mapped when a function returns.

Mapping a memory region without arrays is straightforward. Essentially, memory locations which share a base pointer form fields of a data structure rooted at this pointer, and on the stack, memory locations rooted at the beginning of a function frame represent local variables and function arguments.

When a potential array is detected, we check if it matches the data structure pattern derived from the base pointers. If not, the array hypothesis is discarded. For example, if base pointers hint at a structure with variable length fields, while the presumed array has fields of 4B, DDE assumes the accesses are due to functions like `memset`.

The analysis may find multiple interleaving arrays (each corresponding to its own data structure access pattern indicated by base pointers). If such arrays are not included in one another, we merge them. Otherwise, we examine the base pointers further to see if the arrays are nested.

3. RESULTS

DDE can analyse any application that runs on the Linux guest on our (QEMU-based) emulator. To verify its accuracy, we compare the results to the actual data structures in the programs. This is not entirely trivial. We cannot compare to the original source code since aggressive compiler optimizations may change the binary significantly [3]. Also, DDE cannot discover variables that always remain unused, but this probably should not count as a 'missed' data structure.

We first applied the analysis to a host of handwritten programs as well as to the UNIX `fortune` program. We managed to exercise 71% of all `fortune`'s functions, which variables account for 77% of all variables used by the application. On stack, DDE analyzed correctly 83% of variables (which account for 82% of stack memory). Further, 9% of variables were unused (which account for 13% of stack memory), and in the case of 6% of variables (1% in terms of memory usage) structure fields were classified as separate fields and not as belonging to one structure. (We explained reasons for that in Figure 1.) In the case of heap, we can observe less unused variables: 90% of memory was analyzed accurately, and 9% was again flattened - an 88-byte `FILEDESC` structure contains a nested 12-byte `STRFILE` structure which was not spotted.

We also analyse DDE's performance for other, more complex binaries, these include the Linux loader `ld-2.9.so`, the `Apache` web server, `wget`, `grep`, `gzip`, `glines`, `binutils`, and `gnometris`. Further evaluation can be found in [13].

Limitations. DDE is not flawless. While evaluating it we identified the following limitations.

1. DDE cannot recognise nested structs if the inner struct is never accessed separately. In that case, DDE just returns a single large structure. As the result is equivalent, we do not consider this a problem.
2. DDE does not detect arrays if a loop is executed less than 4 times (in that case it is classified as a structure).
3. DDE cannot classify fields that the program never accesses.
4. DDE cannot currently deal with custom memory allocators. If the program allocates a pool which serves various data structures, and is reused in the runtime, DDE does not handle that correctly. DDE does not detect the custom allocation routine, and thus it gets confused with the interleaving data structures.

Note that even if DDE cannot classify an array or structure correctly in one particular loop or function, it may still get it right eventually. Often data structures are accessed in more than one function, yielding multiple loops to analyse the layout.

4. RELATED WORK

Most existing approaches to decompilation build on static analysis of binaries. The most advanced techniques in this field include value set analysis (VSA) [1], and aggregate data structure inspection (ASI) [12]. The culmination of these static techniques is a combination of VSA and ASI [2].

The idea of ASI originates in efforts to deal with the Y2K problem in old COBOL programs. Translated into C terms, ASI attempts to partition memory chunks statically in `structs` of arrays and variables. For instance, if a stack frame holds 40 bytes for local variables, and the program reads 4 bytes at offset 8 in the range, ASI classifies the 40 bytes as a sturct with one 4-byte variable wedged between 2 arrays. As more addresses are referenced. ASI eventually obtains an approximate mapping of variable-like locations.

ASI has also a clever trick to identify data structures and types, and that is to use the type information from system calls and well-known library functions. As the arguments of these calls are known, at every such call, ASI tags the arguments with the corresponding types and propagates these tags through the (static) analysis.

At this point, however, we have to mention that all of these static techniques have problems handling even the most basic aggregate data structures, like arrays. Nor can they handle other common programming cases. For instance, if a C struct is copied using a function like memcpy, it will be misclassified as having many fields of

4 bytes (on 32 bit machines) simply because the stride in `memcpy` on 32 bit machine is 4). Similarly, they cannot deal with functions like 'alloca'. Finally [2] is context-sensitive, which leads to state space explosion. The reported results show that even the most trivial programs take an exceedingly long time to analyse.

Another system that uses dynamic analysis for data structure recovery is Laika [8], which uses dynamic analysis for data structure recovery. It employs Bayesian unsupervised learning to detect data structures. However, its detection is very imprecise and limits itself to aggregates. For instance, it may observe chunks of bytes in what looks like a list, but it does not know about fields in structures. For debugging and reverse engineering, this is wholly insufficient. The authors are aware of this and use Laika only to estimate, in an approximate manner, the similarity of malware.

REWARDS [11] builds on the part of ASI that propagate type information from known parameter types (of system calls and library functions). Unlike ASI, however, it does so dynamically, during program execution. All data structures that are used in, or derived from, systems calls or known templates are correctly identified. However, REWARDS is (fundamentally) not capable of detecting data structures that are internal to the program.

DDE emphatically does not need any known type to recover data structures, but whenever such information is available, we can take advantage of it to recover *semanatics*. For instance, it might help to recognize a structure as a `sock_addr` structure.

5. CONCLUSIONS

We have described a new technique, known as DDE, for extracting data structures from binaries dynamically without access to source code or symbol tables, by observing how program access memory during execution. As until now data structure extraction for C binaries was not possible, we expect DDE to be valuable for the fields of debugging, reverse engineering, and security.

Acknowledgments

This work is sponsored by the EU FP7 Wombat and the EU FP7 SysSec projects.

6. REFERENCES

[1] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 binary executables. In *Proc. Conf. on Compiler Construction (CC)*, April 2004.

[2] G. Balakrishnan and T. Reps. DIVINE: Discovering variables in executables. In *Proc. Conf. on Verification Model Checking and Abstract Interpretation (VMCAI)*, January 2007.

[3] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What you see is not what you execute. In *In Verified Software: Theories, Tools, Experiments*, page 1603, 2007.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[5] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[7] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Paris, France, April 2010.

[8] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008*, pages 255–266, San Diego, CA, December 2008.

[9] DataRescue. High level constructs width IDA Pro. `http://www.hex-rays.com/idapro/datastruct/datastruct.pdf`, 2005.

[10] M. V. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. *Working Conference on Reverse Engineering*, 0:27–36, 2004.

[11] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, March 2010.

[12] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 1999.

[13] A. Slowinska, T. Stancescu, and H. Bos. Give me back my datastructures! - excavating data structures from stripped binaries. Technical Report IR-CS-57, Vrije Universiteit Amsterdam, May 2010.

[14] A. Slowinska, T. Stancescu, and H. Bos. Precise data structure excavation. Technical Report IR-CS-55, Vrije Universiteit Amsterdam, February 2010.