dynStruct: An Automatic Reverse Engineering Tool for Structure Recovery and Memory Use Analysis

Daniel Mercier
University of Kent, UK
ampotos@gmail.com

Aziem Chawdhary
University of Kent, UK
A.A.Chawdhary@kent.ac.uk

Richard Jones
University of Kent, UK
R.E.Jones@kent.ac.uk

***Abstract*—dynStruct is an open source structure recovery tool for x86 binaries. It uses dynamic binary instrumentation to record information about memory accesses, which is then processed off-line to recover structures created and used by the binary. It provides a powerful web interface which not only displays the raw data and the recovered structures but also allows this information to be explored and manually edited. dynStruct is an effective tool for analyzing programs as complex as `emacs`. A demonstration video is available at: http://bit.ly/2gQu26e**

I. Introduction

Reverse engineering is the process of understanding the behaviour and logic of a program without access to its source code. Reverse engineers are routinely employed by government and commercial organisations to ensure programs are compliant with software licensing, security policies and to identify exploitable vulnerabilities and backdoors [1]. The key goal of reverse engineering is to extract enough information from a program to understand its behaviour: such information can include memory usage, data structures created and manipulated, network usage and interaction with the host operating system via system calls. A typical task for a reverse engineer is to understand the control flow and memory usage of an application and thus identify how the program behaves. However, much of the work undertaken by a reverse engineer is manual and time consuming. Reverse engineers routinely use tools such as debuggers, disassemblers and profiling tools.

dynStruct is an open source tool that helps recover data structures created and used by an executable binary. Identifying data structures is a key task undertaken by many reverse engineers. However, structure recovery is challenging for a number of reasons: compilation can remove useful information such as unexported symbol information, type and size information of variables and structures. In addition to the problems of recovering structures, the control flow of a program is difficult to determine through static analysis. Data structures are commonly used in different ways in different parts of the program, hence understanding control flow is crucial to correctly identifying the data structures manipulated during program execution.

dynStruct uses dynamic analysis and thus avoids the need to recover the control flow of a program before performing useful analyses: this is in contrast to static analysis which requires the computation of control flow information prior to further analysis. By using dynamic analysis, dynStruct can also attempt to analyse obfuscated programs which is a major hurdle for static analysis based tools [2]. Since dynStruct is focused on data structure recovery, it uses dynamic binary instrumentation (DBI) to gather information about memory allocation and accesses. DBI generates vast amounts of data, making manual inspection infeasible. Thus our tool processes information offline before presenting it to the user. Offline analysis reconstructs C-like structures and arrays, and determines where they are allocated and accessed. Engineers can use dynStruct's powerful web-based interface to explore recovered data structures to discover how structures and their fields are accessed.

Our goal is to support real world reverse engineering, e.g. in capture-the-flag security contests (ctftime.org/ctf-wtf), where both speed and accuracy of structure recovery are important. Full details of dynStruct can be found in Mercier's thesis [3].

II. Related Work

We highlight notable work on recovering data structures from binaries. Laika [4] is a Bayesian unsupervised learning system that recovers structures from memory dumps. It is sufficiently accurate for use in malware analysis but dynStruct requires greater accuracy to aid program understanding.

Rewards [5] instruments memory allocations with type attributes which it propagates during analysis. Information from standard library and system calls helps determine the types used by the program. Rewards fails to recover data structures if there is no interaction with known libraries or system calls.

Howard [6] identifies root pointers of data structures from memory allocation routines and statically allocated data. Arrays and structure member types are detected at run-time by searching for specific memory access patterns. Howard claims around 90% accuracy for heap structures and 80% for stack structures. In contrast, dynStruct does not use access patterns but instead records the size of each memory access for offline processing. This allows dynStruct to recover structures accurate sized from only one run of the program. In contrast, Howard needs multiple runs to obtain good accuracy, and its use of the KLEE execution engine [7] leads to a heavy overhead.

TIE [8] uses both static analysis to recover functions and function boundaries, and dynamic analysis of memory accesses to determine the positions of structure members. Its constraint solver recovers the types of structures and their fields. dynStruct does not need to recover function boundaries and records

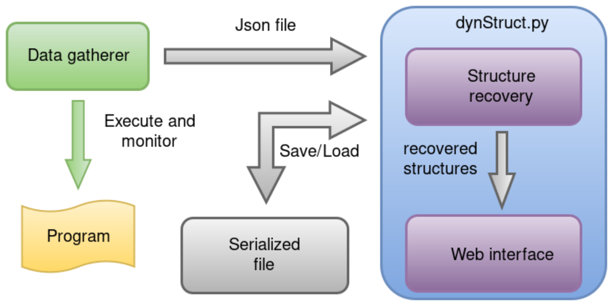


Figure 1. Overview of dynStruct

contextual information for every memory access allowing more accurate type recovery.

Robbins [9] shows how to mathematically characterise the correctness of types recovered from a binary executable, using constraint solving to decompile a binary to a semantically equivalent type-safe C-like 'witness' program, thereby giving confidence that the recovered types are both correct and meaningful. The work has been applied to an idealised x86 language.

Grammatech [10] use a sophisticated static type constraint system to recover types from stripped binaries built on-top of their proprietary CodeSurfer analysis tool.

Unlike dynStruct, none of these tools are publicly available.

III. Data Gathering

dynStruct consists of a data gatherer phase followed by an offline structure recovery phase and presents the results to the user via a web interface (Fig. 1). The data gatherer, written in C, uses the DynamoRIO [11] DBI framework to record every dynamic memory allocation and every access to these allocations, along with some contextual information, and saves this to a JSON file (Fig. 2). DynamoRIO was chosen because of its portability (multi-OS and multi-architecture) and because it is an open source project.

A. Allocation Monitoring

The data gatherer records information about dynamically allocated memory: each call to malloc, realloc, calloc or free is wrapped by pre- and post-call instrumentation. Pre-call instrumentation records parameters used for each allocation and deallocation. Post-call instrumentation records return values. dynStruct can support other memory allocation routines but the names must be the same as those provided by `libc`; we intend to address this in future work. For performance, dynStruct ignores allocations in library routines by default, but the user can provide a list of libraries to be instrumented.

dynStruct terms an allocated region of memory a *block*. A block is *active* until it is deallocated. When a block becomes inactive it is moved from an AVL tree of active blocks to a linked list which stores only inactive blocks. This handles multiple allocations of the same memory region natively, without time-stamps. Block information includes start and end addresses, size, whether the block was freed, the location of the wrapped call, etc.

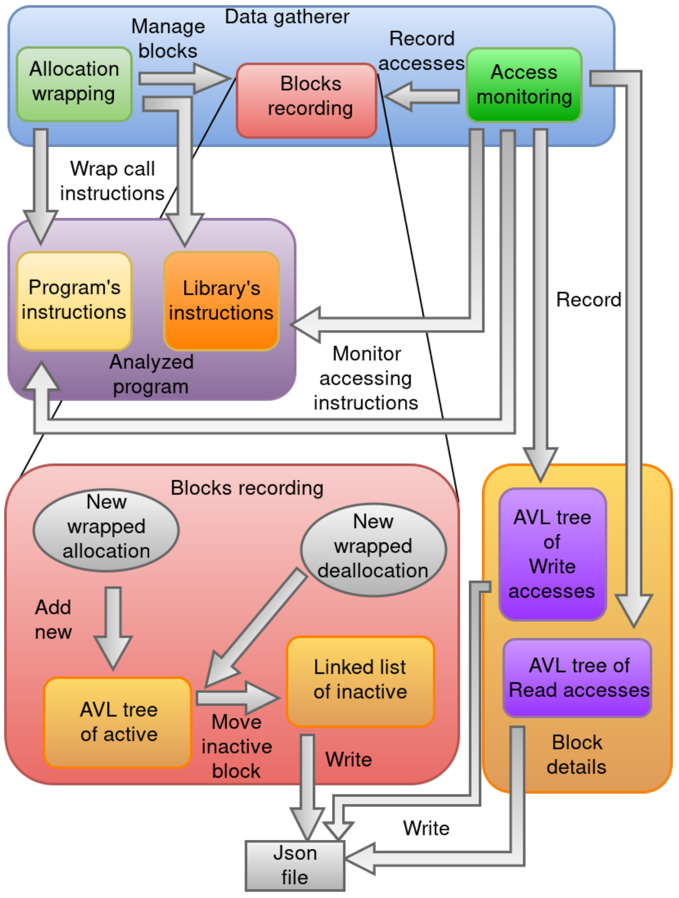


Figure 2. Data gatherer architecture with exploded view of block recording

B. Access Monitoring

Every memory access executed by the program is instrumented. First, we check whether the address accessed is within an active block. If so, the access is recorded and linked to this block. The record includes the size of the load/store, the number of times this access occurred, its offset within the block, and information about the accessing instruction such as the opcode and the opcode(s) of the *context instruction* (see III-D).

C. Function Calls

For every interaction with memory, dynStruct records the current function being executed, which dynamoRIO does not provide. To overcome this, dynStruct records function addresses in a stack. Function names are recovered via a hashmap of every loaded symbol. To identify addresses of external functions, dynStruct identifies the address of the target function by reading the corresponding GOT section, which is detected at runtime to handle position independent code.

D. Context

For each memory access two instructions are recorded: the accessing instruction and a context instruction. The intention is to assist type recovery in the structure recovery phase. For a write, the previous instruction may provide information about how the data was generated. For a read, the following instruction may provide information on how the data is used. Contextual information is important for self-modifying code, which may generate instructions dynamically. Recording these two instructions avoids complex analysis in the structure recovery phase. Instrumentation is performed at the granularity of basic blocks. Consequently, context instructions are not available for writes at the start of a basic block or for reads at the end. Nevertheless, because a member is typically read and written many times, context instructions are usually available for all structure members.

E. Data Recording and Output

dynStruct uses the memory management functions provided by DynamoRIO to separate its data from the instrumented program's. By using a chunked, linked list of 4 KiB pages, allocated on demand, dynStruct can run complex programs under the data gatherer with feasible time and memory overhead. Data is output every time the inactive blocks list reaches 100 blocks long. This keeps memory overhead low even for long running programs. When program execution terminates any remaining blocks, active or inactive, are written to the output file. For example, without this optimization dynStruct was unable to start an Emacs process with than 45K lines of Lisp (configuration files and modules): after 15 minutes the data gatherer stopped because it used more than 2 GiB of RAM. With the optimization, the same Emacs process and configuration started in 6 minutes and used a maximum of only 400 MiB.

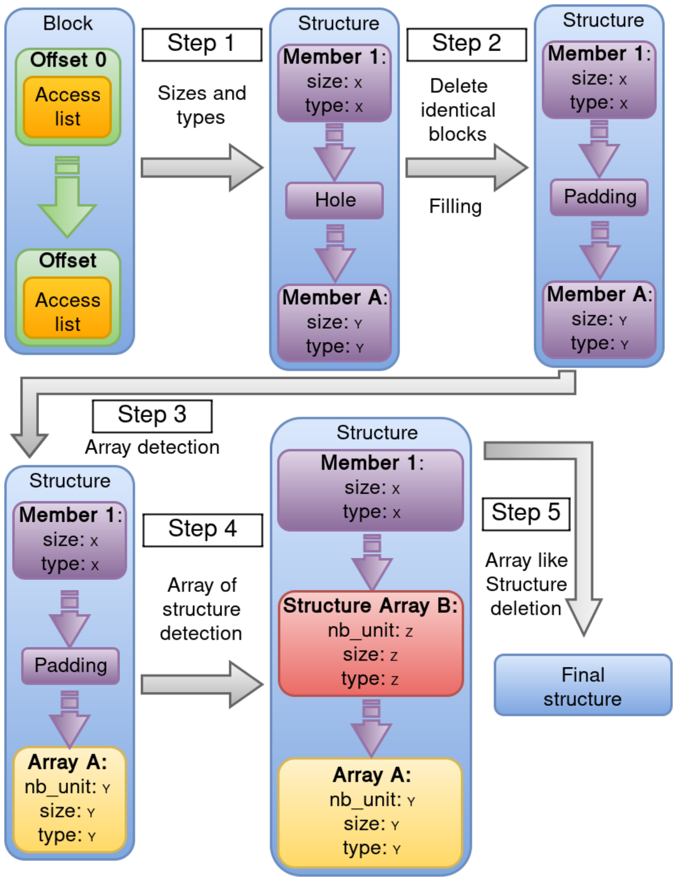


Figure 3. Structure recovery process

IV. Structure Recovery

Initially, every memory block is considered a structure but, over five steps (Fig. 3), anything that does not look like a structure is removed.

A. Step 1: Recover Member Types and Sizes

The first step is to analyse the raw data from the JSON file to find the type of every member of every structure. This step is split in two sub-steps: get the size of the access, then recover the type.

a) Recovering Member Sizes: A member may be accessed with multiple sizes, often because of initialization with memset or compiler optimizations, so recovering the real size of each member is the most important step of structure recovery. dynStruct uses the heuristic that a member's size is the size most commonly used to access it. If this distribution of sizes is multi-modal, the smallest size is used. This mainly happens because of string manipulation with XMM registers.

b) Recovering the Type: The type of a member is determined from the accessing and context instructions. The opcodes recorded by the data gatherer are disassembled using Capstone [12], a universal disassembly framework. Every access is typed, and the most frequent type is retained as the type of the member. The analysis for a write/read is based on checking whether the previous/next instruction respectively provides information that can be exploited to detect its type.

c) Block Comparison: Usually multiple instances of a structure are allocated during the execution of a program. To avoid repeating the recovery process on every instance of the same structure, dynStruct compares the types and sizes of members, block by block. If one of the two blocks has a "hole" instead of a member, the two blocks are still considered as instances of the same structure (because some instances can have unaccessed members). There are only two cases where types can be different but two blocks are still considered as instances of the same structure: if one block has a default type of a pointer size and the other has a pointer or a pointer to function, or if one block has a simple pointer and the other a pointer to function. If two blocks are deemed to be instances of the same structure, they are merged. The merge fills holes with members of the second block when available, and replaces less meaningful types (default type of pointer size and simple pointer) by more meaningful ones (pointer or pointer to function).

B. Step 2: Fill with Padding

At this stage there may be "holes" in recovered structures. Because the accesses are retrieved from only one execution of the program, some access paths may not have been used: without accesses, dynStruct cannot recover members. The compiler may also add padding to satisfy alignment requirements. dynStruct fills these holes with padding, arrays of `uint8_t`.

C. Step 3: Array Detection

So far, only individual members have been recovered, but merging consecutive members which have the same type into an array can increase readability. dynStruct simply replaces a sufficiently long sequence of members of the same type with an array of the same size; the type of the array is that of its members. As it is common for a structure to have a few members of the same type — for example, a coordinate may be a pair of integers — dynStruct assumes that arrays have at least 5 members. Even if this assumption is wrong, the structure's layout will still be correct. It would be possible to remove this restriction by examining access patterns in the data gatherer to reveal inner structures.

D. Step 4: Detecting Arrays of Structures

Similarly, consecutive structures are replaced by an array of structures. Detecting this pattern requires multiple passes until no new array is detected. This allows the discovery of arrays of structures where the inner structure contains another array of structures, etc.

E. Step 5: Fusing Array-like Structures

This last step removes every structure considered to be an array. A structure is considered to be an array if all its members are of the same recovered type or padding. This step can be disabled with a run-time option, as necessary. For structure recovery, showing (typically, many) arrays can obscure important structures in the flow of information. But, for memory use analysis, retaining arrays provides important information about how memory is used.

F. Output

The recovered structures can be written to a file or on the console with a C header style. It is also possible to serialize the recovered structures, loaded blocks and accesses. This allows starting the web interface later directly by loading this serialized file without having to re-run the recovery process.

V. Web Interface

dynStruct's web interface provides a powerful and easy to use tool, linking the raw data and structures recovered to allow a reverser to explore memory usage. A web interface has the advantage of portability and allows collaborative exploration. Data can be obtained either directly from the data gatherer (via the JSON file) and analyser or, more quickly, load from a serialized file.

To help the reverser find data quickly, it is presented in tables, which can be sorted using any column as the key (for example, access size, or the name of a function that called malloc). Fig. 4 shows an example. Rows can be filtered using search boxes in each column. As well as displaying the data in an effective way, the web interface needs to be reactive, even with hundreds of thousands or more recorded accesses. Processing that amount of data in the browser (in JavaScript) is clumsy; for 700,000 entries any sorting or filtering action would take more than twenty seconds. To avoid that, all the sorting and filtering is done in Python before sending the data to the web interface.

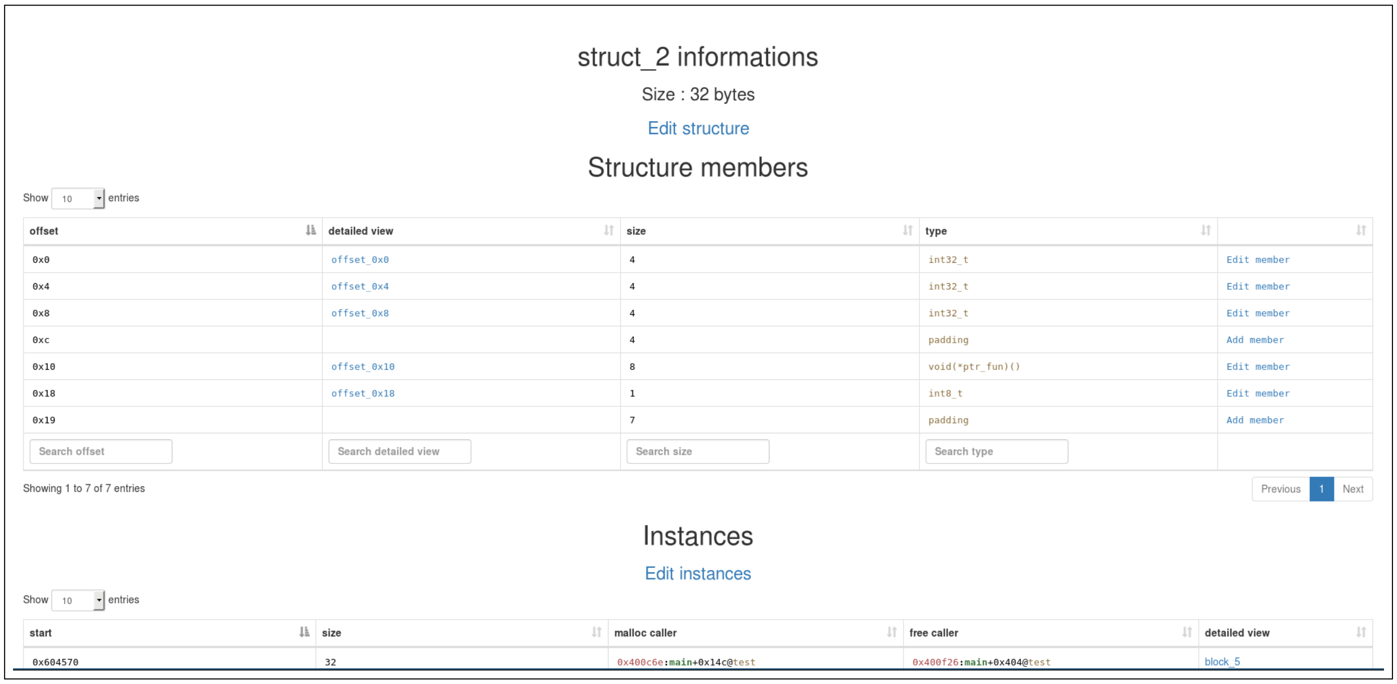
The web interface allows structures to be edited easily, e.g. if a reverser wants to rename a structure member or change its type to improve readability or capture some semantic knowledge. The reverser can also add or remove members, or modify their size. The size of a complete structure cannot be changed, but new ones can be created and existing ones removed. This can be useful where structure recovery considered two similar blocks to be instances of the same structure, but the reverser wants to separate them as they are semantically different. The only condition for adding a block as instance of a structure is that it is the same size as the structure's other instances. All the modifications made in the web interface are automatically saved in the serialized file.

VI. Results

We measured dynStruct's accuracy using a suite of small and large programs. Small programs present a tougher test for structure recovery since its accuracy depends on how many times and in how many places a structure is used. 'Real' programs were used to measure performance and memory overhead. All the tests were performed on a freshly setup VMware virtual machine running 64-bit Ubuntu 16.04 (kernel 4.4), with 4GiB RAM and 2 processors. The only packages installed were those needed by dynStruct and the test program.

dynStruct's overheads are not related simply to the size of the instrumented program but to the number of allocations made and how these are accessed. The data gatherer's memory overhead varied between 4× for emacs (Tab. I) and 20× for small programs, much of which is due to DynamoRIO's and dynStruct's libraries. The performance overhead varies between 20× and 50× that of the original program. The cost of structure recovery depends on the size of the JSON file but dynStruct's performance certainly makes it a useful tool for small and medium sized programs — one of the authors has used it successfully in real time in capture-the-flag contests. Obviously, the cost of data gathering and structure recovery may be expensive for long running programs, especially if a specific action has to be taken to trigger the behaviour to analyse.

We examined the accuracy of structure recovery against the suite of programs use by Robbins [9], excluding those that did not allocate structures. dynStruct's accuracy is good but not perfect, correctly recovering 20/22 structures. It matched 48/61 members exactly and the remaining 13 partially, i.e. with the correct size but not the wrong type. In one example, an array of pointers was recovered as a mixture of `int64_t` and pointer types because not every element had the same context. In another, two semantically distinct but similar structures were recognised as instances of the same structure. dynStruct also misrecovered a structure consisting of a sequence of identical types as an array.



struct_2 informations

Size : 32 bytes

Edit structure

Structure members

Show 10 entries

offset	detailed view	size	type	
0x0	offset_0x0	4	int32_t	Edit member
0x4	offset_0x4	4	int32_t	Edit member
0x8	offset_0x8	4	int32_t	Edit member
0xc		4	padding	Add member
0x10	offset_0x10	8	void(*ptr_fun)()	Edit member
0x18	offset_0x18	1	int8_t	Edit member
0x19		7	padding	Add member
Search offset	Search detailed view	Search size	Search type	

Showing 1 to 7 of 7 entries

Instances

Edit instances

Show 10 entries

start	size	malloc caller	free caller	detailed view
0x604570	32	0x400c6e:main+0x14c@test	0x400f26:main+0x404@test	block_5

Figure 4. Web interface: structure view

Table I

OVERHEAD RESULTS FOR THE DATA GATHERER AND THE RECOVERY PROCESS (ARITHMETIC MEAN AND 95% CONFIDENCE INTERVALS). MEMORY INCLUDES LIBRARY USAGE FOR THE PROGRAM, DYNAMORIO AND DYNSTRUCT. PERFORMANCE OVERHEAD INCLUDES THE TIME TO LOAD AND INITIALISE DYNAMORIO, ABOUT 0.05S

	Data gatherer				Structure recovery		
	memory usage		time		gatherer	memory	recovery
program	original	dynStruct	original	dynStruct	output	used	time
ls	2.4MiB	42MiB	<0.01s	0.16s±0.02s	204KiB	28.5MiB	2.08s±0.14s
netstat	2.6MiB	42MiB	0.016s±0.01	0.33s±0.03s	6.6KiB	23.5MiB	0.18s±0s
emacs -q	27MiB	103.7MiB	0.20s±0.04s	56.59s±2.93	129MiB	2.9GiB	4h30
xterm -e ’exit’	11MiB	68MiB	0.12s±0.01s	4.22s±0.07s	28MiB	900MiB	1h16±23.56

VII. CONCLUSION

dynStruct is a reverse engineering tool which can successfully recover structures used by a program. Its powerful web interface allows a reverse engineer to explore the raw data gathered and the structures recovered. It has been included in WeakerThan Linux 7, a custom security oriented Linux distribution (http://www.weaknetlabs.com/2016/07/wt7-updater-stable.html. dynStruct is available at https://github.com/ampotos/dynStruct.

REFERENCES

[1] J. Baldwin, A. Teh, E. Baniassad, D. Van Rooy, and Y. Coady, “Requirements for tools for comprehending highly specialized assembly language code and how to elicit these requirements,” *Requirements Engineering*, vol. 21, no. 1, pp. 131–159, 2016.

[2] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, “Protecting software through obfuscation: Can it keep pace with progress in code analysis?” *ACM Comput. Surv.*, vol. 49, no. 1, pp. 4:1–4:37, July 2016.

[3] D. Mercier, “dynStruct: An automatic reverse engineering tool for structure recovery and memory use analysis,” Master’s thesis, University of Kent, Nov. 2016. [Online]. Available: http://kar.kent.ac.uk/58461/

[4] A. Cozzie, F. Stratton, H. Xue, and S. King, “Digging for data structures.” in *Operating Systems Design and Implementation*, 2008, pp. 255–266.

[5] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Information Security Symposium (CERIAS)*, 2010, p. 5.

[6] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures.” in *Network and Distributed System Security Symposium (NDSS)*, 2011.

[7] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Operating Systems Design and Implementation*. USENIX, 2008, pp. 209–224.

[8] J. Lee, T. Avgerinos, and D. Brumley, “Tie: Principled reverse engineering of types in binary programs,” in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2011.

[9] E. Robbins, A. King, and T. Schrijvers, “From minx to minc: semantics-driven decompilation of recursive datatypes,” in *Principles of Programming Languages (POPL)*. ACM, 2016, pp. 191–203.

[10] M. Noonan, A. Loginov, and D. Cok, “Polymorphic type inference for machine code,” in *Programming Languages Design and Implementation (PLDI)*. ACM, 2016, pp. 27–41.

[11] D. Bruening, Q. Zhao, and S. Amarasinghe, “Transparent dynamic instrumentation,” in *Virtual Execution Environments (VEE)*. ACM, 2012, pp. 133–144.

[12] “Capstone engine,” http://www.capstone-engine.org.