POSTER: Identifying Dynamic Data Structures in Malware

Thomas Rupprecht[*]
University of Bamberg,
Germany

Xi Chen
Vrije Universiteit Amsterdam,
The Netherlands

David H. White
University of Bamberg,
Germany

Jan Tobias Mühlberg
iMinds-DistriNet, KU Leuven,
Belgium

Herbert Bos
Vrije Universiteit Amsterdam,
The Netherlands

Gerald Lüttgen
University of Bamberg,
Germany

ABSTRACT

As the complexity of malware grows, so does the necessity of employing program structuring mechanisms during development. While control flow structuring is often obfuscated, the dynamic data structures employed by the program are typically untouched. We report on work in progress that exploits this weakness to identify *dynamic data structures* present in malware samples for the purposes of aiding *reverse engineering* and constructing *malware signatures*, which may be employed for *malware classification.*

Using a prototype implementation, which combines the type recovery tool *Howard* and the identification tool *Data Structure Investigator* (DSI), we analyze data structures from *Carberp* and *AgoBot* malware. Identifying their data structures illustrates a challenging problem. To tackle this, we propose a new *type recovery* for binaries based on machine learning, which uses Howard's types to guide the search and DSI's memory abstraction for hypothesis evaluation.

CCS Concepts

•**Software and its engineering → Software reverse engineering; Data types and structures;**

Keywords

Data structure identification, malware, reverse engineering, program signatures

1. INTRODUCTION

As the scope and applicability of malware increases, so does its complexity and, in turn, the time necessary to understand a malware sample via reverse engineering. This is due to the fact that such samples are often heavily obfuscated. However, the increased size and complexity that complicates reverse engineering also mandates the proper practices of code and data organization during development, e.g., by employing functions and data structures. As noted in [5], while the leakage of information due to the organization of code is often mitigated by common obfuscation techniques, the information leakage from data structures is typically not considered. In this paper we describe work in progress on an analysis that exploits this weakness to identify the *dynamic data structures* present in malware, for the purposes of aiding *reverse engineering* and constructing *malware signatures*, which may be employed for *malware classification.*

Approach. Our prototype is based on *Data Structure Investigator* (DSI) [15], which is a dynamic analysis tool capable of identifying (cyclic) singly/doubly linked lists (SLL/DLLs), trees, skip lists and combinations thereof such as nested lists. DSI's original use case was as a program comprehension tool for C programs and, thus, its trace recording *C front-end* employs the readily available type information present in C code. However, source code will not be available in typical malware use cases, and therefore, we utilize the type recovery tool *Howard* [14] to obtain the necessary type information. The result is a new *binary front-end* for DSI that allows for the analysis of x86 stripped binaries, i.e., those with all symbols not required for execution removed.

Novelty & Related Work. DSI provides two desirable features that will help us to advance the state-of-the-art in identifying dynamic data structures from malware. The first is DSI's *rich heap abstraction* that, in contrast to ARTISTE [3], DDT [7], HeapDbg [11], and MemPick [6], permits the identification of many complex data structure implementation techniques. These range from the generic list implementations employed by the Linux kernel, to highly optimized data structures, and to those created via custom memory allocators. Secondly, DSI identifies data structures by *accumulating evidence*, a process which is resilient against the transient corrupted shapes that arise due to manipulation operations, which we term *degenerate shapes.* In contrast, other approaches [6, 7] try to exclude degenerate shapes from the analysis, which may be difficult if well-structured interfaces are required [7] and these are obfuscated. Additionally, there is typically limited support for nested data structures, whereas DSI's key features enable the robust identification of arbitrarily deep nesting.

However, DSI's usefulness is significantly reduced when high quality type information is absent, as is the case in stripped binaries. Thus, in contrast to [8, 10], Howard's

[*]Corresponding author: thomas.rupprecht@swt-bamberg.de

ability to identify the types of *nested structs* empowers DSI, as many complex data structure implementation techniques rely on this programming construct.

While the binary analyses mentioned so far consider reverse engineering in general, Laika [5] is designed specifically for the construction of malware signatures based on the data structures in a memory dump, which are found via machine learning. However, no high-level identification is performed beyond the discovery of recursive pointers, which prevents, e.g., distinguishing a binary tree from a DLL.

Lastly, we note that, although uncommon, there exist approaches targeting the obfuscation of data structures [2, 9]. However, even with obfuscation applied, our rich heap abstraction may help to return useful information, as would be the case with the obfuscation of [9]. Of course, obfuscations that target pointers directly [2] remain out of scope.

Applications. DSI's output can benefit the reverse engineering of a malware sample in two ways. Firstly, we may use DSI's rich heap abstraction to guide the *visualization* of program memory. In contrast to the generic graph layout methods used by [1, 11], our DSI-guided layout is more intuitive and includes the ability to view heap manipulations with animation. Secondly, the notion of degenerate shape gives rise to a simple identification of *data structure operations.* By noting the transitions of shape from correct to degenerate and back to correct, we can tag regions of binary code that are responsible for manipulating a specific data structure. In contrast to the operation detection of DDT, this does not require well-structured interface functions, and thus is applicable to malware.

Dynamic data structures in malware are typically found in *reusable components*, such as a VNC server or a TCP/IP stack. Thus, in addition to constructing signatures that characterize the malware as a whole, which could be used for malware classification in the style of Laika, we may also construct signatures that characterize the components. Since the components seem to reoccur across different malware, we expect the signatures will offer a reverse engineer a number of short-cuts when analyzing a new sample. Lastly, we hope to enhance the quality of signatures by including details of any operations used to manipulate the data structures.

Although our primary motivation is malware analysis, DSI in combination with the new binary front-end will have applications in a number of domains, including: informing *formal verification* [12], where the verification process can be aided by automatically generating program annotations for data structure behavior; and *profiling/optimization* [13], where the granularity can be set for a specific data structure.

Contributions. In the following we take a look at the characteristics of a number of malware samples appearing in the real-world and consider their usage of data structures. We then report preliminary results obtained by applying our prototype implementation to examples extracted from *Carberp* and *AgoBot* malware. Through an analysis of these results, we find that, while the combination of Howard and DSI is very promising, the types recovered by Howard require further refinement to fully enable DSI's rich heap abstraction. We propose a solution based on machine learning to tackle this problem.

2. PRELIMINARY RESULTS

We first describe some malware samples that serve to reinforce our theme of reusable components, and then proceed to describe our preliminary results. To determine the ground truth for data structure usage, we employ the source code (in C/C++) available at https://github.com/ytisf/theZoo.

Malware Samples. *Carberp* is designed to target bank accounts and employs the component HVNC to construct a hidden desktop to which one may connect via VNC. The VNC server uses a complex data structure consisting of nested lists. In addition, Carberp employs the component LwIP to construct a hidden TCP/IP stack to hide communication, which makes use of multiple independent lists. LwIP also forms a component of the scareware malware *Rovnix*.

MyDoom is a worm designed to enable remote control by opening a backdoor. It uses an SLL that implements a priority queue to store mail addresses and an SLL of child SLLs to cache DNS MX records. Identical functionality is also present in *HellBot*. Additionally, in the spam botnet *Grum*, we have found partial reuse of the DNS cache.

AgoBot is a modular IRC bot with an IRC proxy component to obfuscate connections, which employs STL lists.

Results. We have applied our prototype to a DLL of child DLLs extracted from HVNC in Carberp (∼800 LOC) and the C++ STL lists present in the IRC proxy of AgoBot (∼100 LOC). The implementations of both data structures are challenging, and DSI's rich heap abstraction is required to provide a correct interpretation.

We first consider the DLLs of HVNC, where, on creation, a single *memory chunk* is allocated to hold both the head and tail nodes of the list. Thus, while the structs of the other list nodes are used normally, those for the head and tail are nested into an enclosing struct. Howard correctly identifies the nested tail struct, but fails to discover the nested head struct. This is because Howard recognizes nested structs by observing patterns in the offsets applied to base addresses; however, as the nested head struct resides at the beginning address of the enclosing outer struct, it is never accessed by a unique base address and is not distinguished from the enclosing outer struct. Furthermore, Howard is not able to determine that the middle nodes of the DLL and the tail node are of the same type. Hence, DSI only recognizes DLLs comprised of the middle nodes, since the head effectively does not exist and the tail node is of a different type. This in turn prevents detection of the nesting relationship.

The C++ STL lists employed by AgoBot are cyclic DLLs. The implementation inherits from a base struct providing the DLL linkage to produce a specialized list node encapsulating the list payload. However, the head node of the list remains of the base struct type and, thus, it is essential to identify the base struct that runs through all nodes. Unfortunately, since the base struct resides at the start of the specialized list type, Howard does not find it. As such, DSI detects the DLL without its head node, which prevents the cyclic property from being identified.

In both cases, with correct type information DSI could perform a correct identification. Therefore, the analysis precision is lacking only in the binary front-end.

3. TYPE INFERENCE

The problems reported above are two-fold: firstly, Howard misses nested structs that start at the same address as their enclosing struct, and secondly, some struct types that it infers should be grouped into a single type. The second issue is that of *type merging*, which is a well known problem with a variety of solutions surveyed in [4]. As an example, Howard types a memory chunk by its allocation site and then refines this by grouping types that are all touched by a shared sequence of instructions, as is typical during list traversal. However, no approach is able to handle the correct grouping of types when they appear both in isolation and within an enclosing struct, as seen in HVNC.

Our solution to both problems will be to include DSI's rich heap abstraction in the type inference process. DSI functions by first discovering the atomic building blocks of data structures and then identifying the relationships formed between combinations of them. The building blocks essentially correspond to SLLs, which we term *strands*, and a strand consists of a sequence of *cells*, which are subregions of memory chunks. It is this understanding of data structure nodes in terms of subregions of memory chunks, rather than as whole memory chunks, that allows DSI to handle the tricky implementations observed in Carberp and AgoBot.

To begin with, we will obtain memory chunk sizes from monitoring allocations, and memory locations holding pointers from Howard. Furthermore, we are only interested in discovering the types and locations of cells, i.e., the building blocks of strands. Essentially, we wish to frame type inference as a *machine learning* problem, where a hypothesis is an assignment of types to memory chunks and our evaluation measure is based off DSI's heap abstraction. For example, one measure could be to look for hypotheses that maximize the length of discovered strands, with the intuition being that the best type assignment is the one that allows the discovery of the largest data structures.

However, if we would allow arbitrary assignments of types to memory chunks, then the search space would explode. To make the proposed approach feasible, we will employ Howard's type information to guide the search, where type assignments compatible with Howard's output are preferred, but not enforced. In addition, we will enforce hard requirements for cells, e.g., a memory chunk subregion may be a cell only if it has an incoming pointer to the cell start address and an outgoing pointer originating within the cell (with exceptions made for the first and last cells in a strand). We will also exploit requirements over combinations of cells, since it is not interesting to discover strands of length one.

Ultimately, by allowing cells at arbitrary locations, and then promoting useful type groupings via combinations of cells, we expect the discovery of the common linkage elements that proved so problematic in the above examples to be possible. Furthermore, we would be able to group types over any combination of stack and heap memory, a challenge noted in [4] which currently has no solution. This is necessary in practice to group a stack-allocated head node with the remainder of the heap-allocated list.

4. CONCLUSIONS

We described our vision for identifying dynamic data structures in malware, which is based on the already proven techniques of DSI and Howard. As a first step, we employed Howard to perform the type recovery integral to DSI's rich heap abstraction; however, as evidenced by the complex data structure implementations of the Carberp and AgoBot malware, often the types require further refinement. To remedy this, we proposed a new type inference approach framed as a machine learning problem, which will employ Howard's types to guide the search and DSI's heap abstraction to evaluate hypotheses. The resulting identified data structures will aid reverse engineering and improve malware classification by, e.g., enhancing the signatures of [5].

5. ACKNOWLEDGMENTS

This work is supported in part by DFG grant LU 1748/4-1 and the Research Fund KU Leuven.

6. REFERENCES

[1] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *SOFTVIS 2010*, pages 53–62. ACM, 2010.

[2] S. Bhatkar and R. Sekar. Data Space Randomization. In *DIMVA 2008*, volume 5137 of *LNCS*, pages 1–22. Springer, 2008.

[3] J. Caballero, G. Grieco, M. Marron, Z. Lin, and D. Urbina. ARTISTE: Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions. Technical Report TR-IMDEA-SW-2012-001, IMDEA, Spain, 2012.

[4] J. Caballero and Z. Lin. Type Inference on Executables. *ACM Comput. Surv.*, 48(4):65:1–65, May 2016.

[5] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for Data Structures. In *OSDI 2008*, pages 255–266. USENIX Association, 2008.

[6] I. Haller, A. Slowinska, and H. Bos. Scalable Data Structure Detection and Classification for C/C++ Binaries. *Empirical Softw. Eng.*, pages 1–33, 2015.

[7] C. Jung and N. Clark. DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage. In *MICRO 2009*, pages 56–66. IEEE, 2009.

[8] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS 2011*. The Internet Society, 2011.

[9] Z. Lin, R. D. Riley, and D. Xu. Polymorphing Software by Randomizing Data Structure Layout. In *DIMVA 2009*, volume 5587 of *LNCS*, pages 107–126. Springer, 2009.

[10] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *NDSS 2010*. The Internet Society, 2010.

[11] M. Marron, C. Sanchez, Z. Su, and M. Fähndrich. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Softw. Eng.*, 39(6):774–786, 2013.

[12] J. T. Mühlberg, D. H. White, M. Dodds, G. Lüttgen, and F. Piessens. Learning Assertions to Verify Linked-List Programs. In *SEFM 2015*, volume 9276 of *LNCS*, pages 37–52. Springer, 2015.

[13] E. Raman and D. August. Recursive Data Structure Profiling. In *MSP 2005*, pages 5–14. ACM, 2005.

[14] A. Slowinska, T. Stancescu, and H. Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *NDSS 2011*. The Internet Society, 2011.

[15] D. H. White, T. Rupprecht, and G. Lüttgen. DSI: An Evidence-based Approach to Identify Dynamic Data Structures in C Programs. In *ISSTA 2016*, pages 259–269. ACM, 2016.