dynStruct

An automatic reverse engineering tool for structure recovery and memory use analysis

Daniel Mercier

Kent ID: 15907084

dm486@kent.ac.uk

School of Computing, University of Kent

Friday 9th September, 2016

Abstract

In computer security, reverse engineering is understanding how a program work. It can be used for multiple purposes, like malware analysis or security audit of a program. Reverse engineering is possible even without the source of the program. In this case, knowing what data structures are used by the program is a considerable help. But recovering these structures is difficult and time consuming. Also, at the time of writing, no tool doing this recovery has been publicly released.

This paper introduces dynStruct, an open source structure recovery tool. dynStruct recovers structures in two steps. First a data gatherer executes the program and monitors it. The list of all memory accesses made by the program is written to a Json file. Afterwards a script analyzes this Json file to recover the structures. dynStruct also provides a powerful web interface. This interface, in addition to displaying efficiently the structures and raw data from the data gatherer, links the raw data and the recovered structures to allow a quick and powerful exploitation of all this information.

The tests shows that dynStruct can analyze complex program like emacs or xterm. The tests also show that the recovered structures are similar to the original ones. This ensures dynStruct can provide quick and useful information to help reverse engineers in their task.

Number of words: 12530

Contents

1 Reverse engineering

1.1 Introduction

Reverse engineering is the process of understanding the behaviour and logic of a program. In the case of a program written in a compiled language, the reverse engineering process can be done at two different levels, source code and compiled binary.
This paper focuses on reversing compiled programs without any access to their source code.
Although reverse engineering can be associated with illegal activity like breaking license systems of copyrighted software, it can also be used for multiple legal (and necessary) usages like malware detection, protection and vulnerability research in closed source software.
Reverse engineering can be done statically, by using a disassembler and analyzing the resulting assembly code. Tools like IDA Pro [1] and radare2 [2] are very powerful aids for that approach of reverse engineering.
It is also possible to do reverse engineering dynamically. This is traditionally done by using a debugger to stop the program at a specific point in its execution and analyze its memory. Here again IDA Pro and radare2 can be a powerful help, but also any debugger like GDB [3] on Linux or x64dbg [4] on Windows is useful.
But even today, when reverse engineering is mainly done by hand, tools which can do a specific task automatically like building a call graph provide considerable assistance to the reverse engineer.

This paper presents **dynStruct** [5], a tool to automatically reconstruct structures (memory layouts) used by a compiled program. The background will be described in section 1, followed by descriptions of the implementation of dynStruct itself in section 2. An example of use of dynStruct is given in appendix A. Sections 3 and 4 detail the results and limitations of the actual implementation. Section 5 contains ideas to improve dynStruct in the future.

1.2 Problems for reverse engineering complex programs

The more complex a program is, the more difficult is the reversing. Very complex programs can take months for an entire team to completely be reversed. One of the keys to do this faster is recovering the structures the program uses to manipulate its internal data. Knowing the structures used by the program means understanding its internal representation of data; that also means easily finding where the data of these structures are written or used by the program. With that knowledge it is easier to know when, how and by what a specific behaviour of the program will be triggered.

1.2.1 Compilation is not lossless

The compilation process is not lossless. In fact it can remove all human readable information and only keep the information needed by the processor to execute the program. This means unexported symbols such as function and variable names are removed and replaced by offsets, structures are used as a limited address space where members are offset in this address space without any information about size and type except the semantic use to write and read them. It is also difficult to find structure boundaries when directly looking at the memory used by the program (with a memory dump for example). Laika [6] uses this approach but has less accurate results than the other research presented in section 1.3.
This results in difficulties to understand quickly what is the purpose of a specific function or what kind of data are carried by a specific structure.

1.2.2 Structure recovery

Recovering the structures means knowing what data is used at a specific point in the program. This information helps to quickly understand a complex program. For example in the case of a malware which uses a non-standard protocol to communicate over the network, knowing the structures the malware uses to read the incoming data will quickly reverse that protocol. But recovering structures can be a difficult task.
With a static analysis approach the main problem is that control flow is not clearly visible, and sometimes almost impossible to reconstruct. Function pointers are a real problem here because the called function can be determined at run-time. Also a structure is usually used at multiple locations in the program: the first use (just after its allocation) is the initialization which cannot always be trusted for structure recovery. This is due to the fact that structures are sometimes initialized by

putting the value 0 in every byte (with a memset like function for example). The other use of these structures can be in a completely different function in the program which makes static recovery almost impossible a some cases.
With a dynamic analysis it is possible to know what structure is used for every memory access by looking at the address used. The problem here is that doing it by hand is time consuming. After every memory allocation the address space allocated has to be written somewhere and for every memory access the reverser has to find which address space is used (by looking at which one contains the address). Doing this by hand is not realistic for a program which can have hundreds or more allocations during its execution.

1.3 Existing work

Fortunately for reverse engineers, some research exists to do this structure recovery automatically, allowing them to make their analysis faster. This section describes the existing research on structure recovery in compiled programs.

1.3.1 Laika

Laika [6] uses memory images to try to recover structures from located pointers in the memory dump and addresses pointed to by these pointers. The structures are detected via machine learning. The average accuracy of Laika is around 73%. These results are enough for malware classification using internal structures as signatures (the purpose of Laika). For complex reverse engineering these results can be insufficient. Also having to make a memory image of the program the reverser want to reverse is not always convenient.

1.3.2 Rewards

Rewards [7] is based on PIN [8], a dynamic binary instrumentation framework. The use case of Rewards is mainly memory forensics by typing the memory of a program and allowing easy examination. It can also be used for vulnerability detection. Rewards uses well-known library calls and syscalls to determine types used by a program. When available it will also use debug data to recover internal structures. These structures are propagated backward and forward through the execution. This approach is powerful (around 100% accurate) at determining which data structure a program uses to interact with its environment (library, kernel) but does not recover any information about its internal data structures in the case of a stripped binary.

1.3.3 Howard

Howard [9] uses dynamic binary instrumentation through QEMU [10]) in order to analyze a program during its execution and recover the structures use by this program. Howard identifies root pointers of data structures by looking at allocation routines (malloc, realloc, mmap) for data stored in the heap and analyzes stack frames to get root pointers for stack data. Howard also gets root pointers from statically allocated data (globals and static variables in C) by looking at the addresses used to access these data. By looking at access patterns of the data pointed to by the root pointers, Howard is able to identify arrays and determine the types of members of a structure (including inner structures).
In term of results, Howard claims around 90% of accuracy for heap structures and 80% for stack structures.

1.3.4 TIE

TIE [11] can use both static and dynamic approaches. The static approach applies heuristics to the disassembled program to try to find the functions of the program. The dynamic approach detects dynamically the function executed. The positions of structure members are determined by analyzing the access patterns of the allocated memory. Afterwards TIE creates constraints for every member and tries to solve them to recover the type of structure members.

1.3.5 From MinX to MinC

MinX (abstraction of x86 assembly) and MinC (type safe dialect of C) [12] are two intermediate languages used in this solution. First the assembly code of the program is translated from x86 assembly code to MinX. Then the idea behind this paper is to construct a program in MinC which will make exactly the same changes in memory at the same times as the MinX program. At the end structures are extracted from the MinC program. This solution can recover recursive data structures. According to the results announced, the recovery of structures is done very quickly but the examples are only little textbook programs. In term of accuracy this solution has excellent results.

1.3.6 Common issue

All these publications have the same goal, recovering structures used by a program, but for different purposes (malware classification, reverse engineering). That means not all of them can be used for the purpose of reverse engineering.
Some of these papers do not give sufficient information about their overhead (in particular time and memory overhead) to know if they are usable in a time-limited context like a capture the flag (CTF) contest. A CTF contest is a hacking contest. Two kinds of CTF contest exist, jeopardy or attack-defense. A jeopardy CTF contest is made of multiple challenges which have to be solved as quickly as possible to earn the most points possible. The common categories of challenge are reverse engineering, exploits, cryptography, web vulnerabilities and steganography. In an attack-defense CTF contest each team has to protect its own vulnerable server and attack other teams' servers: every successful attack gives points to the attacking team.
In both cases it's really common to find reverse-engineering tasks which have to be done quickly. Reverse engineering tools which need a significant time to deliver their analysis are not usable in these contests.
Another issue with these publications is none of the described tools are publicly released.

1.4 Contributions

These issues of use in a time limited context and the fact that none of the previously described tools are publicly released are the reasons of dynStruct. My first motivation to start dynStruct was being able to solve CTF reverse engineering challenges faster.
dynStruct is an open-source tool (available at https://github.com/ampotos/dynStruct) released with a MIT license, allowing any kind of use or modification of the tool. The first purpose of dynStruct is automatic recovery of structures via dynamic binary instrumentation of a program. In addition to this recovery dynStruct also records useful information about every memory allocation and every memory access, making it a powerful memory use analysis tool. All these data (including recorded data and structure recovered) can be easily exploited via a powerful web based interface which links all of them together. This interface allows, for example, quickly knowing the address of every instruction which wrote a specific member of a specific structure during the analysis of the program.

1.5 Dynamic binary instrumentation (DBI)

Binary instrumentation means modifying the instructions executed by the program to add, change or remove some behaviours. This can be for searching vulnerabilities as AFL [13] does, to verify the memory is not corrupted like Valgrind [14], to analyze performance, etc. The instrumentation can be done at multiple levels. At source code level, it is done by modifying the source code. For example printing a value for debugging purposes is a sort of instrumentation at code source level. It can also be done at compilation level, as AFL does. At this level it's the compiler itself that modifies the assembly output for the input code source. In the case of AFL it adds instructions which allow afl-fuzz to determine the different possible execution paths in the program. It is also possible to perform the instrumentation on an already compiled program by modifying the executable file [15]. The final level where binary instrumentation can take place is at run-time. This is dynamic binary instrumentation. Changing instructions executed at run-time allows creation of tools which don't need to recompile a program to analyze it. This approach also allows modification of the instrumentation depending on the actual state of the program or at a specific run-time event.

1.5.1 Why chose a dynamic binary instrumentation approach ?

The DBI approach was chosen for the analysis of a running binary for multiple reasons.
First static analysis for structure recovery is complex if it wants to be effective whereas dynamic analysis can just look at what the program does and where, instead of trying to predict it via static analysis. The second reason is classical obfuscation techniques modify the execution flow of the program: a very difficult challenge for a static analysis. But with a dynamic approach it is possible just to look at what instructions are really executed. A dynamic analysis also has no problem analyzing self-modifying code. Using a DBI approach allows powerful and fast automatic dynamic analysis. Because these are not based on the same mechanism as debuggers (ptrace on

Linux), there is no change of context at every analyzed instruction, so the slowdown in execution of the program is limited. The final reason is dynamic binary instrumentation based tools do not have to care about anti-debug techniques. There is some possibility of escaping popular DBI frameworks [16], but these are not commonly used in real world programs compared to anti-debugger techniques (at least at the time of writing).

1.5.2 Dynamic binary instrumentation frameworks

This section quickly presents some available DBI frameworks.

Pin

Pin [8] is one of the most popular DBI frameworks (if not the most popular). It is developed by Intel and supported only on X86 (both 32 and 64 bit) architecture. PIN can run on Linux, OS X, Android and Windows.

DynamoRIO

DynamoRIO [17] is an open-source framework which is supported on X86 (both 32 and 64 bit) and ARM; compatibility for AARCH64 (ARM 64 bit architecture) is work in progress at the time of writing. DynamoRIO runs under Windows, Linux, OS X and Android. This framework was supported by Google via GSOC in 2014 and 2015. [18].

Valgrind

Valgrind [14] is more known for its memcheck tool than for its dynamic binary instrumentation frameworks, but the memcheck tool is based on it. Valgrind DBI is made for heavyweight instrumentation but can be used for any kind of DBI based tool (with less performance than DynamoRIO or PIN).

Frida

Frida [19] is different to the other DBI frameworks described here. Frida injects JavaScript into a native process (instead of native assembly for the others). The result is slower than with the other frameworks. It is more accessible to do dynamic instrumentation specific to one program with Frida because of the use of JavaScript but this framework is not made for heavy instrumentation. This makes it perfect for use in a CTF contest for example but not really usable for writing a general tool.

1.5.3 The choice of DynamoRIO

dynStruct uses DynamoRIO to perform its dynamic binary instrumentation. This choice had multiple reasons. First it is an multi-architecture and multi-OS framework with an universal API for all supported architecture and OS. The second reason is the performance of the framework, which is very close to PIN performance (usually a bit better). Also it is easier to control the performance of a tool with DynamoRIO than with PIN because of the multiple way the instrumentation can be done (inline instrumentation or a call with full context saving for example). The last reason is it is an open-source project with regular updates.

2 dynStruct

2.1 Overview

2.1.1 dynStruct history

During a CTF challenge I did at the start of 2015 I realized that I lost a huge amount of time reconstructing the structures used by the program when doing reverse engineering.
After some research I wasn't able to find any released tool which can do that. A few months later, in April, I discovered dynamic binary instrumentation and decided to start a tool which will do the structure recovery using dynamic binary instrumentation. During the development of dynStruct I realized this tool can do more than just structure recovery and be a tool to analyze the use of memory by a program just by recording additional data. A few months later this project became my MSc project for the University of Kent.

2.1.2 dynStruct goals

dynStruct has two main goals: provide a significant help to a reverse engineer and to be usable in both CTF contests and real world reverse engineering.

Help reverse engineers

dynStruct aims to provide a powerful help to the reverse engineer, saving them time during their analysis. In order to do that dynStruct's goals are to gather relevant data, to recover structures by analyzing these data and to make these data and structures easily exploitable via a web interface.

Accuracy vs usability

To be used in a CTF contest dynStruct must be able to analyze a program fast enough. But to be used against a real world program the accuracy of the resulting structures can be more important than the time spent to get the result. dynStruct aims to be usable in both contexts.
The data gathering is always accurate because it is a recording of what was executed and of the context. So for that component only the speed can be a problem.
For the structure recovery both speed and accuracy are important. This means dynStruct has to find the right balance between accuracy and analysis time.

2.1.3 Differences with existing work

dynStruct differs from existing research in two ways: the goals are different and the approach of the structure recovery is different.

Different goals

The goals of the previous research are malware classification (Laika: 1.3.1) or recovery of structures.
dynStruct also has structure recovery as a goal but it also claims to be a tool for memory use analysis, which is not the case of the other research.

Different approach

dynStruct uses a dynamic approach for the structure recovery. This recovery is based on two components, a data gatherer and an analysis of the data.
Existing research which uses a dynamic approach are Rewards (1.3.2) and Howard (1.3.3). TIE (1.3.4) uses a dynamic approach only to generate a trace of executed instructions; there is no recording of any kind of context during the execution.
Rewards and Howard base their members detection on the access pattern: this means they look at the pattern used to access the structure and determine the size of members from this. There is a problem with this approach. If a member is not accessed or if there is some padding added by the compiler, the access pattern seen during the execution may not reflect the original structure.
To counter that dynStruct is based on the size of memory accesses done on structures. This means every time an access is made to a structure, dynStruct records the size of the access, and uses it to determine the size of every member accessed.

2.1.4 dynStruct components

dynStruct is made of two components, the data gatherer which is a client for DynamoRIO and a python script which does the structure recovery and provides the web interface. Figure 1 provides an overview of the different components of dynStruct.
The end of this section describes the three steps of dynStruct, data gatherer, structure recovery and web interface.

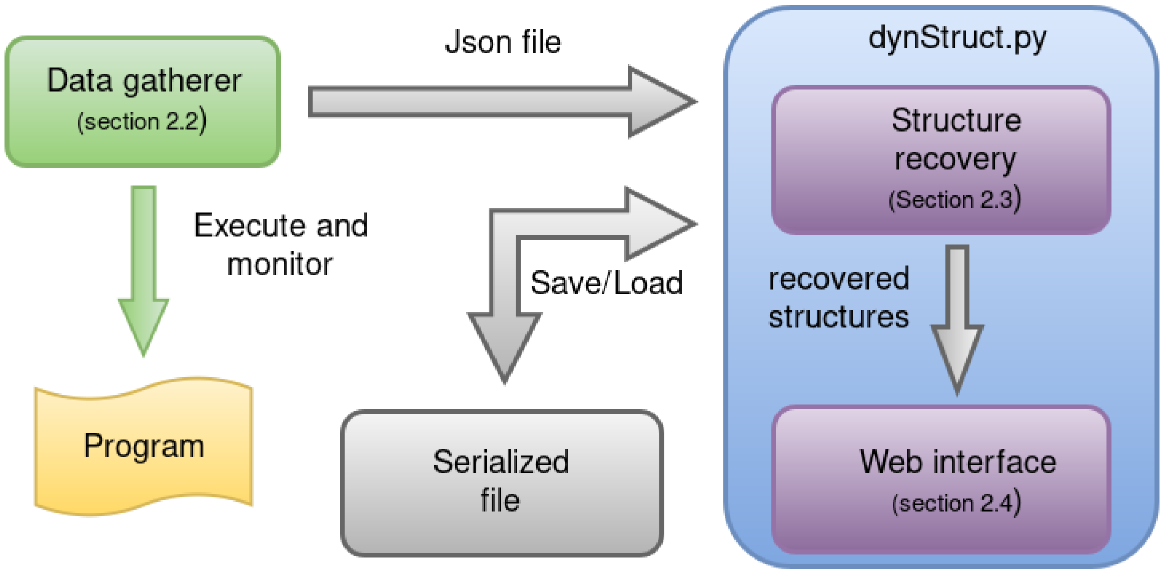


Figure 1: Overview of dynStruct

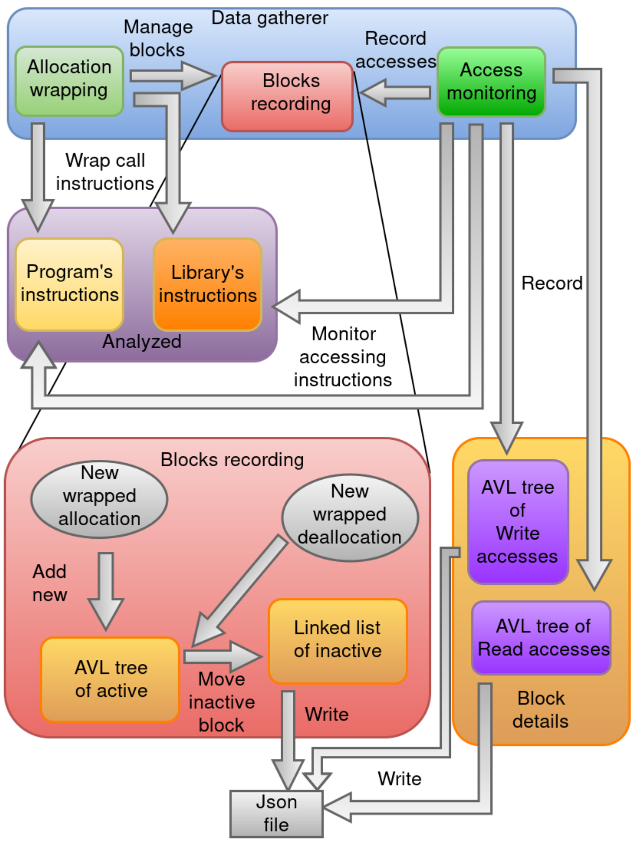


Figure 2: Data gatherer architecture with exploded view of block recording

2.2 Data gatherer

The data gatherer is a client for DynamoRIO. It is written in C. It uses the dynamic instrumentation framework to record data for every memory dynamically allocated and every access on the allocated memory. This component also records some additional data which provides a context to the memory access. This data gatherer writes all the recorded data into a Json file.
An overview of the data gatherer architecture is given in figure 2. Each part of this architecture is describe in this section.

2.2.1 Allocation monitoring

The first thing the data gatherer records is the dynamically allocated memory. To do that every call to malloc, realloc, calloc and free is wrapped by pre-call instrumentation and a post-call instrumentation.
The pre-call instrumentation records the parameters which are the requested size for malloc, calloc and realloc, the address for free and realloc.
The post-call instrumentation records the return value, the address of the memory allocated for malloc, calloc and realloc. Free has no post-instrumentation because the function doesn't return anything.
An allocated memory is called a block in dynStruct; every information which describes a block is stored on a structure. A block is called active if its memory is not deallocated (by a call to free). When a block becomes inactive it is moved from the binary tree containing active blocks (see 2.2.5)

to a linked list which stores only inactive blocks. This handles multiple allocations of the same address space during the execution of the program without using a timestamp.
The information stored into a block are its start and end addresses, size, indication if the block was freed or not and if it was allocated or freed by realloc. Also information about where the call to the wrapped functions occurred is recorded (2.2.3).

2.2.2 Memory access recording

Every executed instruction of the analyzed program which does a memory access is instrumented.
The first thing this instrumentation does is check if the accessed memory is in an active block. If it is not the case the normal execution of the program is resumed. If it is the case the access information is stored in a structure and linked with the active block which contains the accessed address.
The recorded data are the size, the number of times this access occurred, the offset of the access in the block address space, information about the instruction (2.2.3) which does the access, the opcode of the instruction (2.2.3) and the opcode of the context instruction (2.2.3).

2.2.3 Other recording

To do the structure recovery the previously described recordings provide enough data. But by recording additional data and displaying them in the web interface dynStruct allows the reverser to understand all the context of the concerned access or structure member.
These data allow the reverser to easily find a specific access instruction in the disassembly of a program by providing information like the address of this instruction, and the module where this address is located (2.2.3).
Another important information recorded is the instruction which does the access and a second instruction called the context instruction. The analysis of these two instructions allows dynStruct to recover the type of the member such as pointer, function pointer, floating point number and signed/unsigned number. These two instructions also provide a context which can be difficult to have in the case of self-modifying code.

Addresses
Address information is recorded for every instruction which does a recorded memory access and for every call instruction which allocates or deallocates a recorded block. An example of format used for these addresses is ***0x40057a:main+0xd@a.out***.
The first part (before the ":") is the exact address of the instruction during the execution.
The second part (between the ":" and the "+") is the name of the function executing the instruction. If the name is not available (in the case of a stripped binary) the address where the function starts is display instead.
The third part (between the "+" and the "@") is the offset of the instruction in the function.
And the last part (after the "@") is the module containing the function. A module can be the program itself or any dynamically linked library.
In order to know in which function (and the address of start of this function) the execution is in when recording an address, dynStruct keeps a stack. The address of the start of the called function is pushed at every call instruction and the top address in the stack is popped at every ret. Because sometimes a call is used without a ret and a ret without a previous call, some call and/or ret are ignored. When an address is recorded dynStruct gets the address of the start of the current function and looks in a hashmap (which stores all the symbols of the the program and of every loaded library) if a name exists for this function.
On Linux, library calls are in reality a call to a specific section of the program (called PLT). This section will get the address (by reading the GOT section) of the called library function and jump into it. So to know the name of the function called in the library dynStruct has to check, for every call, if it is a call to the PLT section or not. If it is the case dynStruct sets a flag on its stack to denote the address is on the PLT and when the address is requested it gets the address in the library by reading the GOT section. This 'lazy' resolution of the address is mandatory because in some programs, addresses are written in the GOT section only at the first call to the library function and the instrumentation of dynStruct occurs before this resolution. So for the first call to a library function, at the moment of dynStruct instrumentation, the address of the library function is sometimes not yet written in the GOT section.
DynamoRIO doesn't provide the possibility to get the run-time address space of a specific section of the executed binary. Or this address is not always the same, for example a program compiled with the position independent code option (PIE)

can be loaded at random address in memory for security reasons. To get the address space of the PLT and GOT sections, dynStruct parses the different headers of the ELF file (the file format of program on Linux) and determines in what segment are the sections, and their offsets relative to the start of the segment. DynamoRIO has an API call to get the address space of every segment, dynStruct determines the right segment by comparing its permissions (execute, read and write) and the size to the segment described in the ELF file. After it adds the offsets to the start of the segment and gets the PLT and GOT sections runtime addresses. All the PLT and GOT sections of every module loaded are stored in a similar tree as the one used for storing active blocks, to allow quick access to their information.

Instructions

For each memory access two instructions are recorded, the instruction which does the access and a context instruction. This has two purposes. First, these instructions will be analyzed during the structure recovery to recover member's type. Secondly, in the case of self-modifying code, these instructions may be dynamically generated by the program, making them difficult to obtain for the reverser. In this context recording these two instructions allows the reverser to know these instructions without having to do complex analysis to obtain the generated instructions. The context instruction is not the same in the case of a read access and in the case of a write access.
For a write access the context instruction is the previous one, because the previous instruction can provide information about where the data written comes from.
For a read access the context instruction is the next one, because it can provide information on the use of the data stored.
Sometimes the context instruction is not available because the instrumentation is made with a basic block granularity. This means if a write instruction occurs at the start of a basic block or if a read access occurs at the end of a basic block, dynStruct cannot record a context instruction. Even if some access doesn't have a context instruction, because a member of a structure is usually read and written more than once during the execution, every member usually has multiple accesses with context instructions to analyze.

2.2.4 Custom memory allocator for dynStruct recording

To store all the data recorded dynStruct has to do memory allocation, but doing this through a classical call to malloc can break the transparency provided by DynamoRIO. So dynStruct uses the memory management functions provide by DynamoRIO. There are different kind of allocation possible via DynamoRIO, using a managed heap (malloc like allocation), unmanaged using a memory page (separate from analyzed program or not). DynStruct has mainly 2 kinds of allocation for its data, the allocation of the access and all the others. The others' allocation needs a managed heap because they are allocated and deallocated multiple times during the execution of a program. The allocation for the recording of the accesses are different: once allocated they will be deallocated only when a block will be written on the output. Since dynStruct can write the data of the free block on the output during the execution of the program (2.2.6), the deallocation time of access recording is the same as the block where the access was made. Using a managed heap is useful when allocation and deallocation can happen at different moments in the execution, but comes with an overhead in memory (metadata) and in time (the managing algorithm).
So to reduce its overhead dynStruct has a custom allocator for access data. This allocator allocates entire memory pages (4096 bytes) and uses them as an array of structures which store access data. When the array is full another page is allocated. The only metadata present are a pointer to the previous page and the index of the next free structure in that page. When the data of the block are written, at the end of the output all pages linked with this block are freed. This custom allocator is used for the two internal structures storing access data and also for storing the instruction and the context instruction (2.2.3).
The introduction of this optimization allows dynStruct to run complex programs under the data gatherer with a reduced memory overhead and the time overhead was also reduced. For example before this optimization dynStruct was unable to start an Emacs with a configuration of more than 45K lines of code: after 15 minutes the data gatherer stopped because it used more than 2Go of RAM. After this optimization the same Emacs with the same configuration starts in 6 minutes with a maximum usage of only 400Mo.
This significant difference also shows that the Dy-

namoRIO managed heap algorithm has difficulty working with a highly fragmented heap.
The allocation used to get memory pages doesn't separate them from the executed program because this functionality is not yet implemented for Linux. This makes dynStruct visible for the executed program until this feature is implemented.

2.2.5 AVL tree implementation

Having a limited time overhead in the data gatherer is important, especially when executing complex programs. dynStruct needs to keep a quick access to every active block to store access data. If a simple linked list can be enough for little program with just a few allocations, when many hundreds of blocks are active at the same time a linked list is too slow. To keep the access to every active block dynStruct has a custom AVL tree.
An AVL tree is a self balancing binary tree; the complexity to add, access and remove a node in this tree are all the same: O(log n). This allows dynStruct to manage and access the active blocks with a limited overhead. The implementation of the AVL tree used by dynStruct is custom because instead of having a single item for each node it has a set of item (the address space of the block). This means dynStruct cannot handle overlapping blocks, which is not possible unless if the heap of the program is corrupted.
To avoid any problem when using the '-a' option to wrap allocation function provide by an other library than the libc (for example ld-linux.so), the avl tree discard any duplicate. This mean if you ask to add a node which has a starting address already used as a starting address in the tree we don't save it. This could still be a problem in the case of overlapping blocks but overlapping blocks are unlikely to appear on any allocator implementation because of the memory corruption this will entail. However in case of memory corruption due to a bug or a vulnerability in the program this can happen and dynStruct will have undefined behavior (which can be a segfault).

2.2.6 Output

The data gatherer records information for dynStruct but does not analyze them, that part is handle by the script dynStruct.py. So the data gatherer needs to pass the data to dynStruct.py.
To do so the data gatherer writes a Json file with all the recorded data. Json format was chosen for mainly two reasons. First this format can easily represent the data gathered. The second reason is dynStruct is an open source tool, this means it can be used and modified by users. The Json format is supported by almost every language today (natively or via library/module), which makes the use of the data gathered for another purpose than dynStruct simple.
Json format also allows users to get the information needed by the dynStruct.py script by another means than the data gatherer (to use it from an architecture other than x86 or an OS other than Linux for example) without having any issue with the format of the output file. The Json output is not written at the end of the execution of the program but every time the inactive blocks linked list is 100 blocks long. This maintains a low memory overhead even for long running programs. At the end the blocks which are still active and the blocks in the list (if any) are written.

2.2.7 Multi-process programs

By default DynamoRIO loads every child process with a new instance of the client used in the parent process. This allows dynStruct to natively handle multi-process programs. But this native handling had a problem: the output file.
dynStruct has an option to specify the name of the output file, but the client arguments are also duplicated when a child process is created. This made children processes' output files overwritten by others and sometime produced invalid outputs because every process uses the same output filename. To resolve this issue a simple solution was used; the file with the specified filename is used if the file doesn't exist. So now every output file for child processes uses the default format which is ***<program_name>.<pid_of_process>***. In addition a new option which allows choosing the output directory for every output file generated by the program and the child processes was added.

2.2.8 Options

The data gatherer has multiple options, which can be displayed by using the -h option. Some of them are already described, like the outputs options (2.2.6 and 2.2.7).
The option -a was briefly described in 2.2.5. This option allows handling other allocation management routines than the ones provided by libc, but the name must be the same (malloc, calloc, real-

loc and free). Improving the handling of alternative allocation management routines is described in the future work section 5.5

Another option is the possibility to wrap modules, the -w option. By default dynStruct only wraps the call to allocation routine coming from the program itself and ignores the call if it comes from a library. This is done for optimization reason. Because the instrumentation of dynStruct is heavy, this option allows specifying which library has to be instrumented. In addition to wrapping the program itself, dynStruct also wraps every library specified via -w option. This option can be used multiple times.

The last option of the data gatherer is the -m option. This option works exactly as the -w option but enable the monitoring of the module instead of the wrapping. This means the memory access made by specified modules will be recorded.

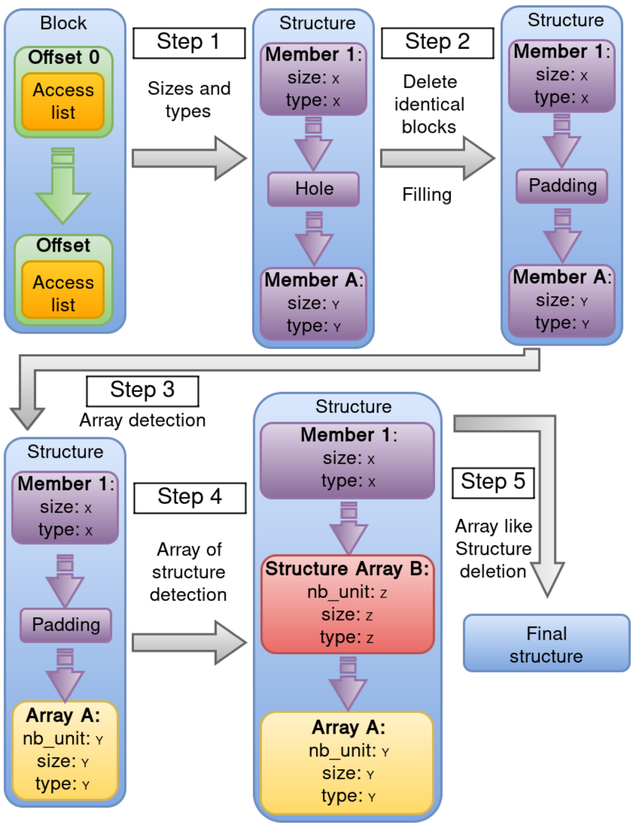


Figure 3: Structure recovery process

2.3 Structure recovery

The structure recovery process is done by the dynStruct.py script which is written in python. This process is made of five steps which are described here (see figure 3).

This process first considers every block as a structure and removes the blocks which don't look like a structure in step 5 (2.3.6)

2.3.1 Step 1: recover members' type and size

The first step of the structure recovery is to analyze the data from the data gatherer. The raw data is loaded from the Json file and analyzed during this step.

The goal of this step is to find the type of every member of structure. This step is split in two sub-steps. First, get the size of the access, then recover the type.

Recovering the size

The first thing done during the structure recovery process is getting the size of every member of the structure. A member can be accessed with multiple sizes, usually due to initialization (memset like function) or compiler optimization, so recover the real size of the member is the most important step of this structure recovery. That is simple to do, just keep the most used size to access a specific offset of the structure. After getting the size of a member dynStruct will determine the offset of the next member by adding the size to the offset of the current member. Because it is possible to have holes in the structure (see 2.3.3, the offset previously calculating may have no recorded accesses. If it is the case dynStruct searches for the next offset with recorded accesses. If multiple sizes are all used the same amount of time, and this amount is the most used sizes, the smallest one is used. This mainly happen with strings because the libc function uses XMM registers to manipulate up to 16 bytes in one instruction.

Recovering the type

After determining the size of the member its precise type will be recovered. This sub-step is based on the analysis of the accessing and context instructions (2.2.3). To analyze these instructions, the opcodes recorded by the data gatherer are disassembled using Capstone [20], a universal disassembly framework. Every access will be typed, and the type with the most occurrences will be kept as the type of the member. Read accesses and write ones don't have the same analysis.

The analysis for write accesses is based on the previous instruction. The idea is to check if this previous instruction provides information exploitable to detect the type of the value written by the access. dynStruct perform three analysis in case of

write accesses. Is the value written a pointer, a floating point value or an unsigned value ?
The floating point analysis is simple: check if the value written is from an XMM register. If it is the case, check the size of the access. For a 4 bytes access the type is **float**, for a 8 bytes access it is a **double**. Sometimes XMM registers are used to manipulate multiple bytes of a string in one instruction; in that case the size of the access is usually 16 bytes long, so dynStruct doesn't type it as a floating point value. Pointer analysis is a bit more complex. If the context instruction is an LEA (load effective address) instruction and if the accessing instruction is a MOV with the destination register of the LEA as source register, the value written is a pointer. If the value loaded by the LEA is RIP relative and the resulting address (instruction address + disp) is on the same memory page as the instruction itself, this pointer is a pointer to a function; if not it is a simple pointer. Every pointer is typed as **void *** except pointers to function typed as **void(*)()**. If the accessing instruction is MOV of a direct value with the access size of a pointer, and this value is an address in the same page as the instruction itself, the access is also typed as a pointer to function. The last analysis for a read access is the detection of unsigned values. The accessing instruction has to be a MOV and the source register has a to be the result of an unsigned operation made by the context instruction.
The analysis for read accesses mostly detects pointers. They can also detect some floating point numbers. These analyses are based on the next instruction.
The first check is if the accessing instruction is a call: the value has to be a pointer to a function. The second check is also simple: if the accessing instruction writes the value read in an XMM register with a size of 4 or 8 bytes, the access is respectively typed as **float** or **double**. If the context instruction is a call using a register written by the accessing instruction, the access is also typed as a pointer to function. The other checks verify if the value read is dereferenced in the context instruction. If it is the case, the access is typed as a pointer. A comment saying pointer to array is added if the dereference is "base + index", another comment saying pointer to structure is added if the dereference is "base + disp". These comments are added because they are typical patterns used by compilers to access members of a structure or values stored in an array.

When no type can be recovered for the member, the default type will be used. The default type is: ***int<number_of_bits>_t***. All members have a default name following the format **offset_<offset_in_the_structure>**

2.3.2 Block comparison

Usually a structure has multiple instances during the execution of a program; this means the structures is allocated more than once. To avoid performing all the recovery process on every instance of the same structure, a comparison is made between step 1 and 2.
This comparison just checks if the type and size corresponds between every member of the two blocks compared. If one of the two blocks has a "hole" instead of a member for an offset, the two block are still considered as instances of the same structure (because some instances can have unaccessed members). There are only two cases where the type can be different and the 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. After this comparison, if the two blocks are instances of the same structure, a merge is performed.
This 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).

2.3.3 Step 2: fill with padding

At this stage there can be some "hole" in recovered structures. These holes can have two origins; they can be unaccessed members or padding to satisfy alignment.
Because the accesses are recorded on only one execution of the program, some functionalities of this program may have been unused. This can lead to unaccessed members in of some structures and without access dynStruct cannot recover these members. The second reason for these holes is padding add by the compiler to satisfy data alignment. For example if a compiler uses an alignment of 8 bytes, this means every structure has to have a size which is a multiple of 8 bytes. So a structure made with a short (2 bytes) and a char (1 byte) will not be 3 bytes long but 8 bytes long with 5 bytes never used.

dynStruct fills these holes with padding. The padding used is an array of uint8_t. The name of every padding array is of the format **pad_offset<offset_in_the_structure>** and the displayed type in the web interface is “padding”.

2.3.4 Step 3: array detection

Up to now only individual members were recovered, but merging consecutive members which have the same type into an array can increase the readability of the structure. This is exactly what dynStruct does in this step 3.
Basically dynStruct just counts how many consecutive members with the same type there are, removes them from the structure and puts an array with number of units equal to the number of removed members. The type of the array is the same as the members. There one exception to this operation. An array has to be made of at least 5 members. If there are only 4 or less consecutive members with the same type, they are not replaced by an array. This is because it is very common to have a few members of the same type following each other in the declaration of a structure. For example a structure made for storing coordinates can have two integers following each other without being an array.
This assumption can be incorrect (a structure can have an array of 2 units). But even if it is the case, the memory layout of the structure will still be correct because two variables following each other and an array of two units are exactly the same from a memory point of view.
It is possible to remove this exception by looking at the access pattern, which can also allow detecting inner structures in a structure. But that has to be added in the data gatherer (see future work 5.3.1).
The name of arrays follows the format **array_<offset_in_the_structure>**.

2.3.5 Step 4: array of structures detection

After array detection, dynStruct detects arrays of structures.
An array of structures is easily detected by a simple pattern detection. Every instance of the detected pattern is a unit of the array. Similar to array detection, every member detected as a member of the array of structures is removed and replaced by the array of structures. The pattern detection does multiple passes until no new pattern is detected. This allows finding arrays of structures where the inner structure contains another array of structures and so on.
The format for the name of array of structure is **struct_array_<offset_in_the_structure>**.

2.3.6 Step 5: deletion of array like structure

This last step is enabled by default but optional; the option -k of dynStruct.py disables it. The idea is to remove everything which does not look like a structure.
This step removes structures which are made only of members with the same type. This includes arrays and simple members; padding is also ignored. For example consider the following recovered structure:

```
struct  my_struct {
float   nb_1;
int8_t  padding[12];
float   nb_array[6]
};
```

The padding is not accessed during the execution of the program. This structure will be consider as an array of float (because there is only float and padding) and will be removed from the structures (except if the -k option is used).
This suppression is made to increase the readability. In a program it is common to have arrays of multiple sizes, but dynStruct will recover them as different structures because of the different sizes. This will create multiple structures (especially when the program use multiple strings) and can make the real structures more difficult to find.

2.3.7 Output

Structure recovery has multiple possible outputs. The recovered structure can be written to a file or on the console with a C header style. It is also possible to serialize the recovered structures and 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. It is also possible to start the web interface directly (with the structures recovered) without writing anything on the console or on files by using the -w option.

2.4 Web interface

The web interface integrated in dynStruct provides a quick way to exploit efficiently the data gathered by dynStruct and the structures recovered. The raw data and recovered structures are linked together to help the analysis. It also allows the reverser to update the recovered structures and to create new ones.
The web interface is provided by the script dynStruct.py by using the option "-h".

2.4.1 Options

By default the interface listens only on the local interface (127.0.0.1) on the port 24242; the option "-l" can be used to change that. Listening on an interface other than the local one allows collaborative analysis of the data and structure display by dynStruct.
The web interface can get its data from the output of the data gatherer, in which case the interface start after of the structure recovery process. If a serialized file is used the web interface starts instantly.
In the case where the reverser doesn't need structure recovery but just the raw data from the data gatherer, the option "-n" can be used.

2.4.2 Why a web interface ?

dynStruct includes an interface to allow easy exploitation of the data recovered and recorded.
An interface is needed because the raw data can be difficult to exploit and the links between the recovered structure and the raw data from the data gatherer are not always obvious. The interface provided automatically links these data to allow easy and powerful exploitation.
The choice of a web interface is not trivial. The fact that this interface is web-based natively allows multiple reversers to look at the same instance of dynStruct interface. This makes reverse engineering in teams more efficient.
Using web technologies for the design, including the CSS framework Bootstrap [21], allowed me to produce an user-friendly and well designed interface quickly and easily.

2.4.3 Displaying data efficiently

An important goal of this interface is to display data efficiently, allowing the reverser to quickly find the data he is looking for.
For that purpose data are presented in tables. In search views every the data can be sorted with any column as key and search box are also present for every column. Also colours are used, making every component of an address information easily identifiable. The format of the displaying address information is: "**<address>:<function_name>+ <offset_in_function>@<module_name>**" (function name if replaced by function start address if not available).

2.4.4 Performance

In addition to displaying the data in an effective way the web interface must be reactive, even if the amount of data available is huge.
Complex programs can have hundreds of thousands or more accesses recorded. Sorting and filtering that amount of data in the browser (in JavaScript) is not feasible; for 700,000 entries any sorting or filtering action would take more than twenty seconds. To avoid that, all the sorting and filtering are done in python before sending the data to the web interface.

2.4.5 Structure editing

The web interface allows to the structure to be edited easily.

Modifying a structure

During its analysis the reverser may need to rename a member of a structure or change its type to a custom one to improve readability or capture some semantic knowledge.
In the structure view of the interface it is possible to edit a structure. Editing allows changing the name of the structure and adding or removing a members. A member can be one of the following: a simple member (for example an int), an array, an inner structure or an array of structures. It is also possible to edit a member, the reverser can modify its size, name and type.
The size of a structure cannot be changed but a new structure can be created (and existing ones can be removed).

Adding or removing instances of a structure

During the recovery process, blocks similar to each other are considered instances of the same structure (see 2.3.2). But two structures can be

similar but don't have the same usage (so the reverser may want to separate them into two structures). It is also possible there is some false positive and false negative in that detection of instances.
To allow the reverser to fix that, it is possible to add and/or remove blocks from the instance list of every structure (even the structures created in the web interface). The only condition on adding a block as instance of a structure is for the block to have the same size as the structure.

Edits saving

All the modifications done in the web interface are automatically saved in the serialized file. This allows the reverser to save his/her work and continue reversing later with its modification on the structures.
If no serialized file is provided to dynStruct.py, the save will not be done.

3 Results

3.1 Test environment

All the results are from tests running on a freshly setup VMware virtual machine with the following specifications: Ubuntu 64 bit 16.04 (kernel 4.4), 4gb of ram (+ 4gb of swap) and 2 processors. The only packages installed are the ones needed by dynStruct and by the programs used for the tests.
Two pools of programs were used for these tests. The first pool is made of little programs and will be used to test the accuracy of the structure recovery. Little programs are used because they use fewer structures than complex programs, making the comparison with the original easier. Even if these programs are small, the results of these tests will be realistic because the accuracy of recovered structures depends how many times this structure is used (and in how many different places in the program). So the recovery will be more accurate on a complex program than on a simple one.
The second pool uses complex programs and some little programs. It was used to test the performance and memory overhead of dynStruct.

3.2 Overheads

The overheads (in performance and memory usage) of dynStruct are not related to the size or the complexity of a program. They depend on how many allocations are done and how many accesses to these allocations are done. Complex program are likely to have a larger overhead than smaller ones but this is not always the case.
The following results show the overhead on tested programs.

3.2.1 Data gatherer

The performance overhead arithmetic mean and confidence interval (with 95% confidence) for ten executions. The memory usage doesn't change between multiple run of the same, so the value is exact. The memory usage reported includes the memory used by all library used by the executed program, so a part of the memory overhead is due to the size of the DynamoRIO and dynStruct library. Also a part of the performance overhead is due to the time taken to load the DynamoRIO run-time and initialize it, about 0.05s
The results are shown in Figure 4.
The results show clearly it is not possible to predict dynStruct overheads. For small programs the memory overhead (x20) seems more important then for emacs (x4) for example. This is due to DynamoRIO and dynStruct libraries being loaded in memory. Because this memory overhead is limited, it will be a problem only on programs already using a huge amount of memory during their executions.
The performance overhead seen is between 20 and 50 times slower than the original program. But this result shows that dynStruct is usable for almost every program even if they are complex. But in the case of some very complex programs which are slow even executing in the normal way, the data gatherer will not be easy to use (especially if a specific action has to be taken to trigger the behaviour to analyze).

3.2.2 dynStruct.py

The figure 5 shows the time and memory taken by dynStruct.py to recover structures. The time reported is also the confidence interval (with 95% confidence) of ten executions of the recovering process. For emacs only one execution was made (because it takes a long time) and for xterm two executions were made.
This result shows that the time and the memory needed to do the recovery is directly related to the size of the Json file from the data gatherer. This file is usually small for little programs, so their

program invocation	memory usage	dynStruct memory usage	original time	dynStruct time
ls	2.4Mo	42Mo	<0.01s	0.16s±0.02s
netstat	2.6Mo	42Mo	0.016s±0.01	0.33s±0.03s
emacs -q	27Mo	103.7Mo	0.20s±0.04s	56.59s±2.93
xterm -e 'exit'	11Mo	68Mo	0.12s±0.01s	4.22s±0.07s

Figure 4: Overhead results for the data gatherer

program	size of data gatherer output	memory used	time to recover
ls	204Ko	28.5Mo	2.08s±0.14s
netstat	6.6Ko	23.5Mo	0.18s±0s
emacs -q	129Mo	2.9Go	4h30
xterm -e	28Mo	900Mo	1h16±23.56

Figure 5: Time and memory used by the recovery process

recovery is fast. For more complex programs the recovery is longer, sometimes a few hours. Also the amount of memory needed on very complex program can be huge. So the structure recovery may be too long or need too much memory to be used on large and complex programs, but it is fully usable for other programs.

3.3 Accuracy of recovered structure

Figure 6 shows the accuracy of the structure recovery against several little programs. The accuracy here is the exactness of the recovered structures and of the type of recovered members. The pool of tests used here is the same as the one used in the MinX to MinC paper [12]. These are programs from a suite of textbook examples [24]. Some of the programs are missing because they don't use any structure.
The comparison of member types is based only on the basic types. This means all typedefs will be ignored and considered as the basic type they replace. Also pointers are always just recovered as "pointer" (with sometimes a hint like "pointer to structure" or "pointer to array" comments), so the pointed type is not used in the type comparison.
For the two "member match" columns, the number of members used is the total number of members of all the structures used by the program. The "exact member match" column shows how many recovered types are an exact match for the recovered member. Similarly the "partial member match" column indicates the the match is only partial. This means the size is the same but not the signedness or the floating point state of the type is wrong. There is no "no match" column because that never happened during the testing.

There are two other mistakes dynStruct did during these tests which are not shown in Figure 6. The first happened when analyzing binomial: one more structure is detected. This structure is suppose to be an array of pointers. But, because every unit of this array didn't have the same context, some are recovered as int64_t and others as pointers. So dynStruct considers this as a structure. The second mistake happened on the analysis of hashsep. Only one structure is recovered but all the blocks allocated by hashsep are detected as instances of this unique structure. This happened because the two structures have a similar pattern (a int followed by a pointer). So for dynStruct the two structures are identical.
When looking at this result it is clear than dynStruct has good results but it is not perfect. The main errors are or type detection for members. These errors are mainly due to a not related context when the members are accessed. This made dynStruct unable to discover these members as a pointers, so it kept the basic type for that size which is int64_t. Another common error of dynStruct is recovering a structure as an; array, this is usually because these structures are made of several members which have same type.
These results show that dynStruct can be trusted by a reverse engineer to provide accurate information about the recovered structures. Even if the types recovered are not always a perfect match compared to the original ones, there is always a partial match which means just the signedness or

program name	structures correctly recovered	structures recovered as arrays	exact member match	partial member match
aatree	1/1	0	4/4	0
avltree	1/1	0	4/4	0
binheap	1/1	0	2/3	1/3
binomial	1/2	1/2	3/5	2/5
hashquad	1/2	1/2	3/3	0
hashsep	2/2	0	2/2	0
kdtree	1/1	0	0	3/3
leftheap	1/1	0	4/4	0
list	1/1	0	2/2	0
mergesort	1/1	0	1/2	1/2
pairheap	1/1	0	2/2	2/2
queue	1/1	0	4/5	1/5
redblack	1/1	0	4/4	0
skip	1/1	0	3/3	0
sort	1/1	0	1/3	2/3
stackar	1/1	0	2/3	1/3
stackli	1/1	0	2/2	0
treap	1/1	0	4/4	0
tree	1/1	0	1/3	2/3

Figure 6: Accuracy results of structure recovery

the floating point state is wrong.

4 Limitations

Each component of dynStruct has specific limitations. Some of them were shown in the results section, some are caused by not yet implemented features and others are due to what a structure can be.

4.1 Data gatherer

The limitations of the data gatherer are unimplemented features and the overhead.

4.1.1 Heap monitoring only

At the time of writing dynStruct only monitors heap allocations. Some ways of storing data rely on dynamic allocation (such as tree, linked list or graph). Also, dynamic allocation is the common way to have structures with a lifetime not related to the stack frame of the function allocating it. But, some structures may only be used on the stack. In that case dynStruct cannot retrieve them. Monitoring the stack to recover the structures used there is described in the future work section (5.4).

4.1.2 No handling of custom allocators

To monitor heap allocation dynStruct wraps allocation routines and records the allocated address space. Even if dynStruct allows wrapping function in a library other than the libc (via the '-a' option) it can only wrap functions named malloc, realloc, calloc and free. Also these functions have to use the same combination of arguments and returned values as the libc ones. In the case of a program using a custom allocator or using raw memory pages (using the mmap syscall, for example) dynStruct is not able to monitor the uses of this allocated memory. A solution to this limitations is described in 5.5.

4.1.3 Overhead

A lot of effort was put in reducing the overhead of the data gatherer, but it is not possible to have an overhead of 0. The overhead of a specific program is not really predictable because it depends on the number of allocations, the number of memory accesses (and how many of the accesses are done in a block). Also using options to specified monitoring ('-m') and wrapping ('-w') can have a huge impact on the overheads. In some cases this overhead may be to important to allow a real analysis of the program, especially if the time is limited. There is some idea in the future work section to reduce these overheads (5.1).

4.1.4 Supports only X86 architectures and Linux

dynStruct is, for now, exclusively developed on Linux and supports only X86 (32 and 64 bit) architectures. This limits the possibility of using it on any other architectures or OS. But porting it is not a too heavy task. As described in section 5.6, dynStruct is mainly written with already portable code.

4.1.5 Path coverage

The data gatherer records block and accesses for only one execution of the program. The only recovered structures are the ones used in this execution. This clearly means the amount of structures (and members of structures) recovered depends on the path coverage of the execution. So it may be necessary to execute the program multiple times with different options or to trigger different behaviours of the program to have all the structures and members recovered.

4.2 Structure recovering

The structure recovering cannot be 100% accurate. This has mainly two causes: members not accessed during the execution and the fact than two different structures in the source can be used in the exact same way in the compiled program.

4.2.1 Unrecovered structure members

When a member is not accessed during the execution of the program via the data gatherer, the structure recovery process will consider it as padding. This makes the recovered structure not identical to the original one. This is due to the path coverage limitation described in section 4.1.5

4.2.2 Accuracy

Even if every member is accessed, the recovered structure may not the same as in the source code. This is due to different types being used in the same way, different structure layouts having the same memory pattern after compilation or not enough context to recover the exact type.

Equivalent types
There are only a few basic types, defined by a size (usually 1, 2, 4 or 8 bytes), if they are signed or not and if they are floating point or integer numbers. An exception for the pointer types (which are basically 4 or 8 bytes long unsigned integers) which have a special purpose : storing memory address. All the other types used are just aliases of a basic type. dynStruct tries to recover the basic type only. So the recovered type may not be the same as in the source of the program but rely on the same basic type.

Equivalent layouts
When looking only at the memory and at what value will be stored in the member, there is no difference between an array of X units of type Y and X consecutive members of type Y. dynStruct doesn't rely on access patterns (yet, see section 5.3.1), so it cannot know the difference and every succession of more than 4 members with the same type will be merged as structure. If there are 4 or less consecutive members, they are considered as consecutive members. This means the result may differ when compared to the original structure but the memory layout of the recovered structure and the one of the original structure will be the same.

Unrelated context instruction
The data gatherer records two instructions, the accessing instruction and a context instructions. The goal here is to extract some context of usage of the stored value. But, sometimes the context instruction is unrelated to the value stored. In that case dynStruct keeps the default type (which is only based on the size of the access). Section 5.3.2 describes a solution to unrelated context instructions.

4.2.3 Memory and time consumption in dynStruct.py

In its actual implementation, the dynStruct.py script keep every block, access and structure in memory during the recovery process and during the use of the web interface. This can be a problem with complex programs. For example Emacs with some configuration and modules installed and without any option used for the data gatherer produces a Json file of more than 1Go. On a computer with 8Go it is not even possible to load the Json file in memory, so recovery is not possible either. To reduce the impact of this limitation a solution is described in section 5.2.
Time consumption can be another limitation. Time consumption is a problem for huge blocks

with thousands of accesses due to the actual implementation of step 1 of the recovery process. Step 1 will search for each offset in all accesses of the block which accesses are done at this offset. For example a Json file of 128Mo, produced by the execution of “emacs -q” (which mean without loading the configuration) takes around 4 hours to go through the structure recovery process. The greater part of that time is used to process only one block, an array of int8_t (maybe a string) of more 89,000 bytes and the block has more than 300,000 accesses. Even if the possibility to have a serialized file allows reloading of a previously recovered program without processing it again is possible, this time consumption is too great. Also the solution proposed for memory consumption will have a positive impact on the performance of the structure recovery for block with a huge amount of accesses (database will be quicker than a list to search in thousands of accesses).

5 Future work

DynStruct is already a powerful tool, but some improvements can be done.
This section describes future improvements which will be done to dynStruct. This includes removing some of the limitations described in section 4 and the addition of new features.

5.1 Data gatherer optimization

The data gatherer is a critical part of dynStruct in terms of optimization because it is the component which execute programs and extracts data from these executions.
In this actual implementation the data gatherer does all its instrumentation via what DynamoRIO calls clean calls. A clean call is an instrumentation based on the call of a specified function with complete context switching. This means all the context of the processor (all registers) are saved and restored after the call. Even if dynStruct does not save the floating point context (because it does not use it), this context switching at every instrumentation is one of the root causes of the performance overhead of the data gatherer.
DynamoRIO provides other types of instrumentation including inline instrumentation. Inline instrumentation will just change the executed instructions in a program without any context save. Using this instrumentation instead of clean calls will reduce the performance overhead but this will be more complex because instructions will have to be manipulated directly. Also for some of the clean calls it may not be possible to change them to inline instrumentation (it depends on the complexity of the instrumentation).
Another possible optimization of the data gatherer is the buffering of the output. Actually dynStruct uses the “dr_printf” function provided by DynamoRIO which doesn’t have buffering. This means a system call will be executed at every call of this printf function and a system call means a context switch to go into kernel land, which is time consuming. Adding a buffering of the output and print into the output file only every 4096 bytes for example (so that a raw memory page can be used for the buffering) will reduce the time used to write data on the output file.

5.2 dynStruct.py optimization

At the time of writing, the dynStruct.py script always keeps in memory every block, access and structure. This is not a problem for little programs but for complex ones the amount of data is too high to be kept in memory. For example a run of emacs with some module and an important configuration can have an output of more than 1Go without monitoring or wrapping any library. To fix that, storing the data in a database like mongodb can be a good idea. This will require more work to remove the serialized file and replace it by an import/export of the database. The goal of this is to reduce the memory used by dynStruct.py and if possible also improve the performance.

5.3 Structure recovery accuracy

The structure recovery is not 100% accurate and it is not possible to reach this accuracy (because two different structures can be the same from a memory point of view). But it is possible improve its accuracy. This improvement can come from more data gathered by the data gatherer or from deeper analysis by the dynStruct.py script.

5.3.1 Via data gatherer

dynStruct does not look at access patterns to recover structures (the approach used by Rewards 1.3.2 and Howard 1.3.3), mainly to be able to handle non accessed members and compiler padding. But using access patterns as a secondary source

of data for structure recovery can help to improve the accuracy of the structure recovery.
Because compilers usually use known patterns to access a structure (first get the root pointer and then add an offset) and array (get the root pointer and add an index), they can be used for better detection of arrays and detection of inner structure. Actually array detection detects if all the members of a structure have the same type at the end of the recovery process (with the padding ignored). Using only this approach can lead to a few false positives and false negatives. But using both approaches to detect arrays can reduce the number of false positives and false negatives. It can also allow removing the minimal size limitation for array (which is 5 units).
Because a specific pattern is usually used to access inner structures, it could be possible possible to detect inner structures during the structure recovery process. This will allow to recover structure in a more closer way to the original than dynStruct today.

5.3.2 Via dynStruct.py

The recovery process is done by dynStruct.py, so it is logical that script can do more to improve the accuracy of this recovery. Detecting the real type of a member is difficult: it requires analyzing the semantics of instructions manipulating the value stored in that member.
Today the data gatherer records one context instruction which is analyzed to recover the type of a member. But sometimes, this context instruction (which is the previous instruction or the next one) is unable to provide any context to the access. In that case dynStruct.py could perform a static analysis on the program itself by disassembling and analyzing the entire function directly from the program file, instead analyzing of only two instructions to get this context. In order to simplify this analysis, the program slicing technique [22] can be used on the disassembled function before analyzing it to only keep the instruction manipulating the read or written value. With this dynStruct will be able to recover precise types for more members than actually.

5.4 Stack analysis

The data gatherer only monitors access on the heap. This means structures only used on the stack will not be recovered.
By looking at the access pattern, the data gatherer would be able to identify structures on the stack. After identifying, just considering it as a block and considering it free on the return of the current function should be enough for dynStruct to handle structures used on the stack as well. The most difficult part will be to know the size of the structure: it may require being able to change the block size after its creation.

5.5 Mmap and custom memory allocator handling

In its current state dynStruct only monitors calls to malloc, calloc, realloc and free. Even if it is possible to wrap these functions in any library via the "-a" option, wrapping other functions as memory allocators and deallocators is not possible.
Becoming more flexible on these allocators will allow dynStruct to recover structures for multiple languages and to be able to analyze programs using their own allocators and deallocators. This can be difficult to implement in a generic way because dynStruct will have to handle different combinations of arguments and return values. Using a configuration file which will define the function name to wrap and describe the combination of arguments and return value can be a good solution. This configuration also should tell the data gatherer where (what arguments) and how (for example in calloc case, two arguments have to be multiple to get the size of the allocation) it can get the data needed to create and remove blocks.

5.6 Ports

dynStruct works only on Linux and X86 (32 and 64 bit) architectures. Porting it for other OS and architectures would be a huge improvement. Fortunately dynStruct is mainly already portable.
The script dynStruct.py is written in python3 making it available for any OS and architectures where python3 is available. The only exception is the analysis of the context instructions analyzes only X86 architectures. But this analysis is using Capstone [20]. Capstone can handle other architectures just by changing a parameter when initializing the disassembler. The code which does the analysis may also need to do specific analyze on each different architectures. All the other components in dynStruct.py can already handle any architecture.

The data gatherer is compiled, so it needs to be recompiled for any new architecture. Also it uses the API provided by DynamoRIO for every interaction with the executed program or the OS. DynamoRIO is portable and actually available for the three main OS (Linux, Windows and OS X): just a few changes (like enabling Windows specific options) are needed in dynStruct's use of the API to be portable. The data gatherer also reads and parses the executable file to recover the address of the GOT and PLT sections (2.2.3). The mechanism behind the GOT and PLT sections is not present in every file format and OS. So to be usable on other OS the data gatherer will have to have a specific handler for library calls for each OS. In terms of architecture, every mechanism of wrapping and recording only uses the API of DynamoRIO, which is the same for every architecture. Some little change may be needed (like monitoring new type of memory access specific for the ARM architecture) to work properly on other architecture, but except that, the data gatherer is already portable. In term of dependencies, the data gatherer doesn't have any, except DynamoRIO. This will also help to port to other OS easily.

6 Conclusion

dynStruct is a reverse engineering tool which can successfully recover structures used by a program. It performs this recovery in two steps, data gathering followed by recovering. dynStruct provide a powerful web interface which allows a reverse engineer to efficiently exploit the raw data gathered for the structure recovery and the recovered structure by linking all of them together.
dynStruct already has users. Github repository statictics shows that dynStruct is cloned, on average, three or four times per week with a peak at more than 40 clones during the first two weeks of July. During these two weeks, dynStruct was also tweeted and re-tweeted many times by well-known accounts in the security community like Binni Shah or Capstone Engine (https://twitter.com/search?f=tweets&vertical=default&q=dynstruct%20since%3A2016-07-04&src=savs). At the time of writing, the dynStruct repository has 110 stars and 16 forks. Several websites like reddit (https://www.reddit.com/r/ReverseEngineering/search?q=dynstruct&restrict_sr=on) and ycombinator (https://hn.algolia.com/?query=dynstruct&sort=byPopularity&prefix&page=0&dateRange=all&type=story) have posted it in their news. WeakerThan Linux 7, a custom security oriented Linux distribution, included dynStruct in its 11th of july update (http://www.weaknetlabs.com/2016/07/wt7-updater-stable.html).

Appendices

A Example of use

This section provides an example of use of dynStruct. For this example a reverse engineering challenge from plaidctf 2016 (available at https://github.com/ctfs/write-ups-2016/tree/master/plaidctf-2016/reversing/bitwise-250) will be solved using dynStruct to analyze its use of memory. The challenge is named `bitwise` and worth 250 points.
This section is not a write-up of the challenge: just the part of the analysis using dynStruct is described step by step. For the other steps required to solve the challenge, just an overview will be provided.
The program asks for a password when it is executed without an argument and writes *"Wrong"* when given an incorrect argument.
This is a classical reverse engineering challenge: the input string which will make the program print a success message will be the flag which unlocks the points during the CTF.

A.1 Spoiler alert

When I did this example I started with no knowledge of how `bitwise` works. But, to help you follow this example I describe here how it works.
`Bitwise` takes an argument and encodes it (I will explain how) and compares the result with a hard-coded value.
The encoder is a Burrows-Wheeler transform [23]. First a squared matrix is created: in each line put the data and rotate it for the next line. For example:

d	a	t	a
a	t	a	d
t	a	d	a
a	d	a	t

Next, the transformation will sort the matrix using the first column as key. The result in our example is:

a	d	a	t
a	t	a	d
d	a	t	a
t	a	d	a

The result of the transformation is the last column (tdaa in our case) and the key is the number of the line containing the original data, 3 in our example. With just the last column it is possible to rebuild the matrix, and the key indicates the line to read to get the data.
`Bitwise` does this transformation on the bit representation of the argument and compares the last column with the hard-coded data (which is the last column of the same transformation applied to the flag).
Please keep in mind that this description of how `bitwise` works is here just to help following the example. The example of use was written when I solved this challenge. I started with no knowledge at all of `bitwise` and discovered things and hints step by step.

A.2 Overview of the example

Here is described every step I took to analyze this program with dynStruct. The detail of each step is in the following text of this section.

Step 1	Data gathering
Step 2	Structures recovering
Step 3	Blocks analysis
Step 4	Accesses analysis
Step 5	Comparison of execution
Step 6	Gather and analyze libc memory use

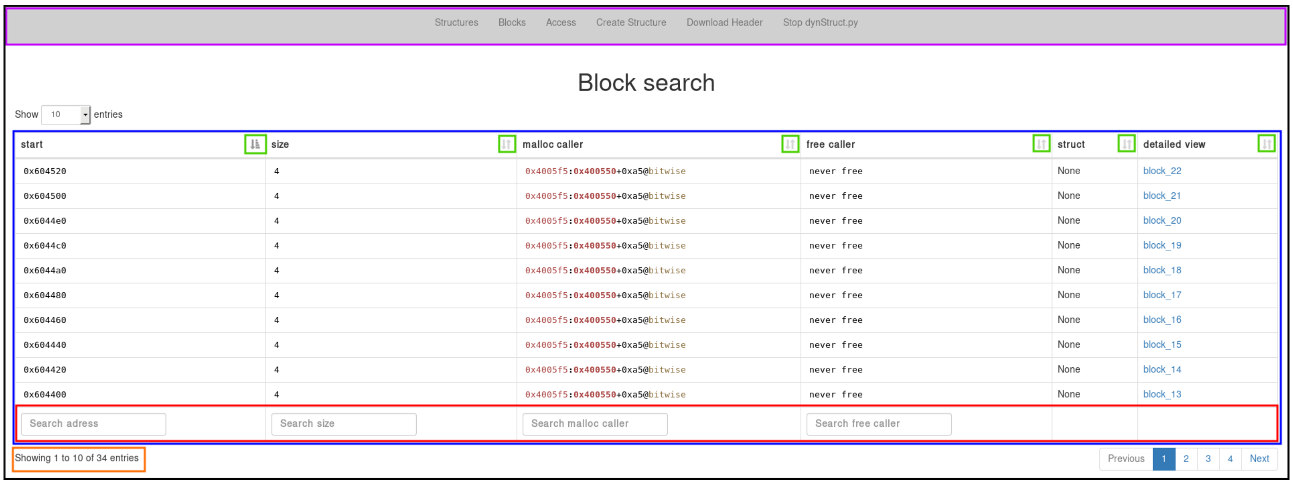
These steps are specific to the analysis of `bitwise`. For another program they may be different but the first four steps will be similar. The other two steps are here to validate or not some thoughts I had during the first four steps.

A.3 Note on screenshots

In all screenshots (except the first one) the blue squares are zoomed areas of the screenshot to increase the readability of important data.
All addresses (instruction addresses for accesses and call addresses for allocation/deallocation) displayed by dynStruct's web interface follow the format: "**<address>:<function_name>+<offset_in_function>@<module_name>**".

A.4 Step 1: Data gathering

The first thing to do is to gather the data with: **"drrun -c dynStruct -o run_1 – ./bitwise 4242"**. This executes `bitwise` with the argument "4242" under the data gatherer. The option "-o run_1" names the output file run_1, this file is 362K long at the end of the execution of the data



Block search

Show 10 entries

start	size	malloc caller	free caller	struct	detailed view
0x604520	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_22
0x604500	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_21
0x6044e0	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_20
0x6044c0	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_19
0x6044a0	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_18
0x604480	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_17
0x604460	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_16
0x604440	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_15
0x604420	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_14
0x604400	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_13
Search adress	Search size	Search malloc caller	Search free caller		

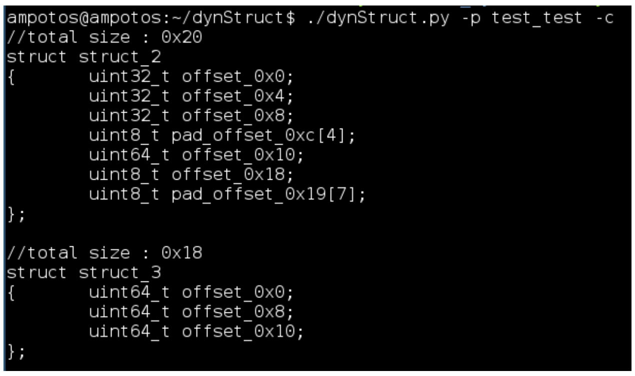
Showing 1 to 10 of 34 entries

Figure A.1: Block search view when starting the interface

gatherer.

A.5 Step 2: Structures recovering

Next we have to recover the structure: **“python3 dynStruct.py -o run_1.serialized -d run_1 -c”**. This takes run_1 as input file and will save the result of the structure recovery (with all blocks and accesses) into the run_1.serialized file. The “-c” option is used to display the recovered structures into the console. The command displays the message *“No structure found”*. This means the program doesn't use structures but may use arrays because there are some blocks and accesses recorded (the output file of the data gatherer is not empty). dynStruct allows analyzing how memory is used even if no structures are used. In contrast, figure A.2 is a screenshot of the result of the -c option with the data gathered on a program which does use structures.



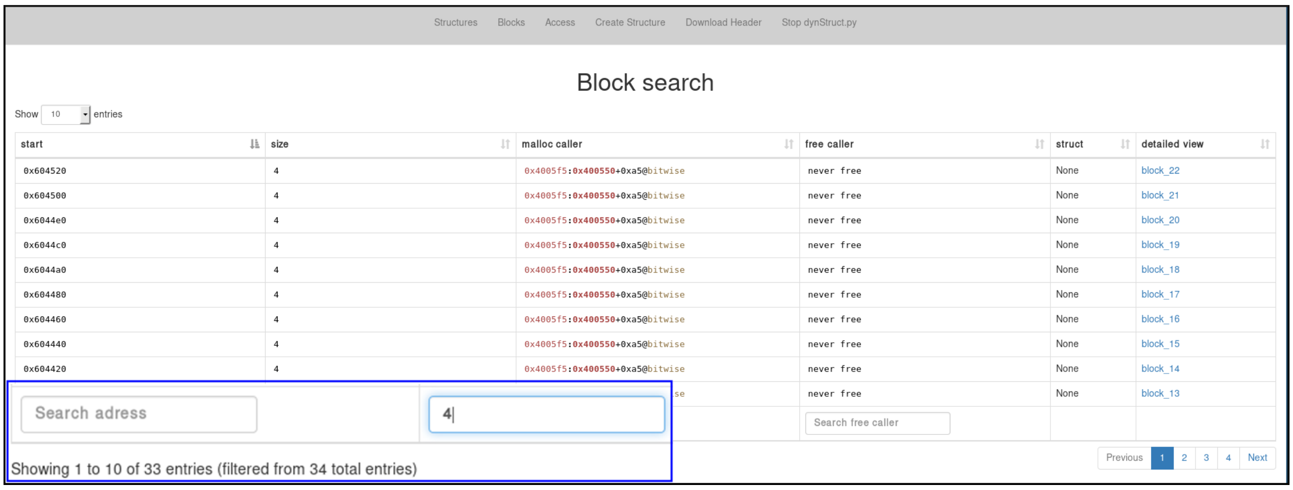
```
ampotos@ampotos:~/dynStruct$ ./dynStruct.py -p test_test -c
//total size : 0x20
struct struct_2
{       uint32_t offset_0x0;
        uint32_t offset_0x4;
        uint32_t offset_0x8;
        uint8_t pad_offset_0xc[4];
        uint64_t offset_0x10;
        uint8_t offset_0x18;
        uint8_t pad_offset_0x19[7];
};

//total size : 0x18
struct struct_3
{       uint64_t offset_0x0;
        uint64_t offset_0x8;
        uint64_t offset_0x10;
};
```

Figure A.2: Example of output of dynStruct.py -c with a program using structure

A.6 Step 3: Blocks analysis

Now to analyze how the program uses its allocated memory, the web interface must be launched.**“python3 dynStruct.py -w -p run_1.serialized”**. This used the serialized file “run_1.serialized” as input and the -w option starts the web interface. The output on the console is: *“Starting web server at 127.0.0.1:24242”*, which means the web interface listens on 127.0.0.1 (only the host can access it) on port 24242. Just copy the given address (“127.0.0.1:24242”) into a web browser to see the web interface.
The web interface starts in the search blocks view (figure A.1). This view displays every recorded block with its information (the blue rectangle in figure A.1 shows where the information is displayed). It also allows searching a specific block with search fields (the red rectangle in figure A.1). The navbar (the purple rectangle in figure A.1) allows navigating through the different views.
By looking at this view we can start to analyze the memory used by `bitwise`. There are 34 allocations (the orange rectangle in figure A.1 shows the total number of blocks), and looking at the first page of the data-table, a lot of them are 4 bytes long. By putting 4 in the search field in the column “size” we can see there are 33 allocations of 4 bytes (see blue square in figure A.3). Now by sorting the data-table by descending size (via the little symbol after the name of the column (green squares in figure A.1)) we can see a 256 byte block.



Block search

Show 10 entries

start	size	malloc caller	free caller	struct	detailed view
0x604520	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_22
0x604500	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_21
0x6044e0	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_20
0x6044c0	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_19
0x6044a0	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_18
0x604480	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_17
0x604460	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_16
0x604440	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_15
0x604420	4	0x4005f5:0x400550+0xa5@bitwise	never free	None	block_14
		se	never free	None	block_13
Search adress	4		Search free caller		

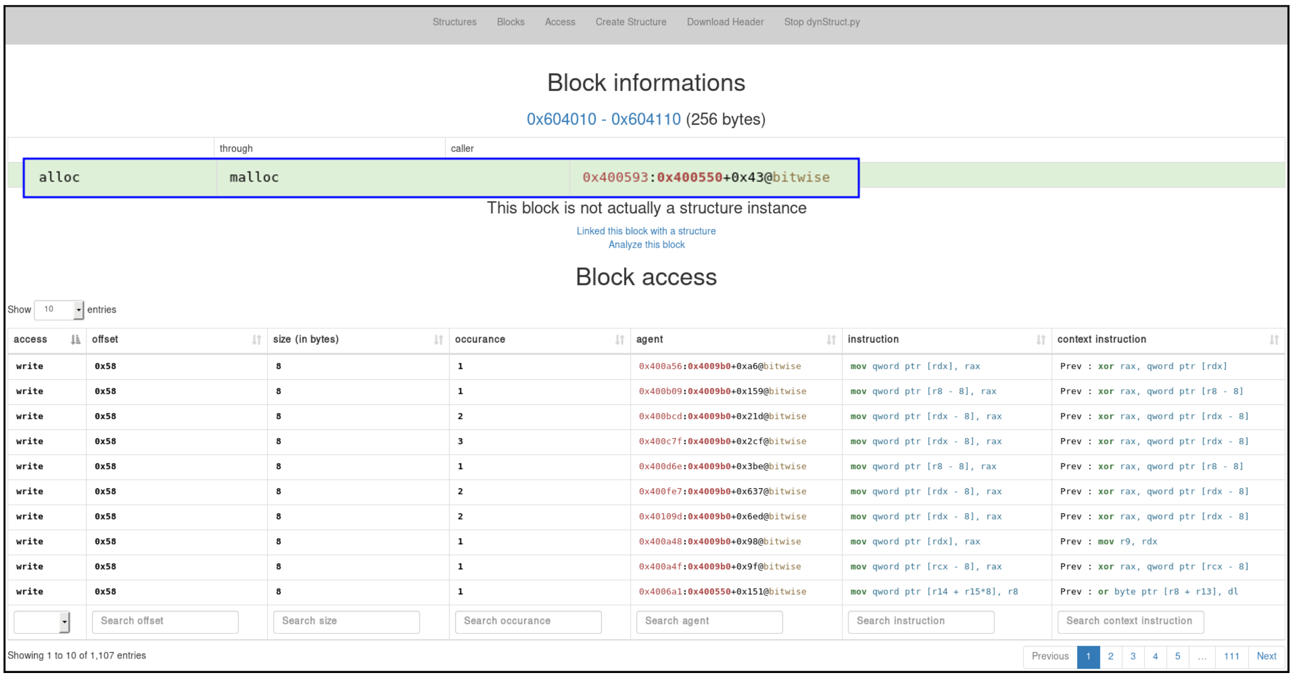
Showing 1 to 10 of 33 entries (filtered from 34 total entries)

Figure A.3: Search for all 4 bytes long blocks

A.7 Step 4: Accesses analysis

The purpose of this step is to find out how the different types of allocation by `bitwise` are used. For that I take a look at the "malloc caller" column. I can see multiple blocks seem to be allocated in the same function starting at 0x400550 (if this function had a symbol, dynStruct would have display it instead of the starting address). We can check if all the blocks are allocated by this function by putting ":0x400550+" in the search field of the "malloc caller" column. Putting ":" before the address and "+" ensures we find exactly this exact address (because of the format used to display the information). After filtering, all the blocks are still displayed, so all the blocks were allocated in this function. I will call this function the "allocating function". Now multiple 4-byte allocations are allocated by the exact same instruction (at 0x4005f5). To verify if all 33 blocks are allocated at the same place, it is possible to put "0x4005f5:" in the "malloc caller" search field. This time only 32 blocks are displayed, so there is one 4-byte block which is allocated somewhere else. I will call the 32 blocks "little blocks" and last one the "lonely block". Clean the search field and sort the data-table by descending malloc caller: the first displayed 4-byte block (block 33) is allocated at 0x4005a3.

Now I know there are 34 allocations, one of 256 bytes and 33 of 4 bytes. I also know all 4-byte allocations are done by the same call to malloc, so it must be in a loop. All the 34 allocations are made in the same function, which start at 0x400550.
To continue this analysis I must go to the detailed view of some of these blocks. I will start with the 256-byte block. To go to its detailed view just click on the link of the "detailed view" corresponding to this block (see figure A.4). dynStruct didn't find any structure pattern in this block so it must be an array. dynStruct can recover the type of the array. For that I just need to click on the link named "Analyze this block". Now the block is linked to "struct_1" (the default name given by dynStruct). Clicking on the link "struct_1" shows me the detailed view of the structure. In this case it is an array, so dynStruct considers it is a structure of only one member. As shown in figure A.5, this member is an array of int64_t (the default type for members of this size). If I click on this single member, named "array_0", I can see that there are 32 units in this array. Looking at the number of units and the size of each unit, this array may be an array of pointers (`bitwise` is a 64bit program) where every pointer points to one of the 32 little blocks (I will check that later by comparing this result with another execution of `bitwise`, in section A.8).
Back on the detailed page of the longest block, I will take a look at the data-table. This data-table contains every access made to this particular block. The first thing I notice is some accesses are made in a function other than the allocating function; this function starts at 0x4009b0. To check if some access is made by the allocating function, I can search for ":0x400550+" in the agent column (see figure A.6). I can see there are 32 read and 32 write accesses made by this function. The write accesses are at 0x4006a1 but the allocation of this block is at 0x400593. These two instructions are very close to each other. This may mean the array is initialized just after its allocation and never written again in the allocating function. The 32 read accesses are made elsewhere in the function,



Block informations

0x604010 - 0x604110 (256 bytes)

	through	caller
alloc	malloc	0x400593:0x400550+0x43@bitwise

This block is not actually a structure instance

Linked this block with a structure
Analyze this block

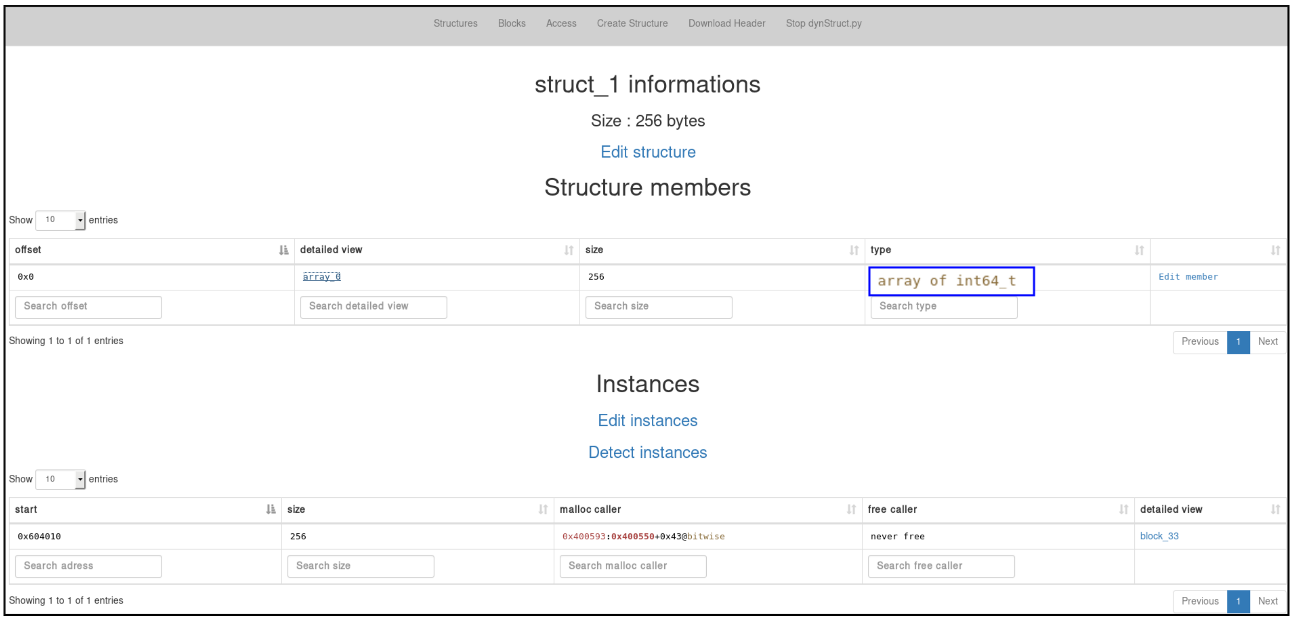
Block access

Show 10 entries

access	offset	size (in bytes)	occurance	agent	instruction	context instruction
write	0x58	8	1	0x400a56:0x4009b0+0xa6@bitwise	mov qword ptr [rdx], rax	Prev : xor rax, qword ptr [rdx]
write	0x58	8	1	0x400b09:0x4009b0+0x159@bitwise	mov qword ptr [r8 - 8], rax	Prev : xor rax, qword ptr [r8 - 8]
write	0x58	8	2	0x400bcd:0x4009b0+0x21d@bitwise	mov qword ptr [rdx - 8], rax	Prev : xor rax, qword ptr [rdx - 8]
write	0x58	8	3	0x400c7f:0x4009b0+0x2cf@bitwise	mov qword ptr [rdx - 8], rax	Prev : xor rax, qword ptr [rdx - 8]
write	0x58	8	1	0x400d6e:0x4009b0+0x3be@bitwise	mov qword ptr [r8 - 8], rax	Prev : xor rax, qword ptr [r8 - 8]
write	0x58	8	2	0x400fe7:0x4009b0+0x637@bitwise	mov qword ptr [rdx - 8], rax	Prev : xor rax, qword ptr [rdx - 8]
write	0x58	8	2	0x40109d:0x4009b0+0x6ed@bitwise	mov qword ptr [rdx - 8], rax	Prev : xor rax, qword ptr [rdx - 8]
write	0x58	8	1	0x400a48:0x4009b0+0x98@bitwise	mov qword ptr [rdx], rax	Prev : mov r9, rdx
write	0x58	8	1	0x400a4f:0x4009b0+0x9f@bitwise	mov qword ptr [rcx - 8], rax	Prev : xor rax, qword ptr [rcx - 8]
write	0x58	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
	Search offset	Search size	Search occurance	Search agent	Search instruction	Search context instruction

Showing 1 to 10 of 1,107 entries

Figure A.4: Detailed view of the longest block



struct_1 informations

Size : 256 bytes

Edit structure

Structure members

Show 10 entries

offset	detailed view	size	type	
0x0	array_0	256	array of int64_t	Edit member
Search offset	Search detailed view	Search size	Search type	

Showing 1 to 1 of 1 entries

Instances

Edit instances

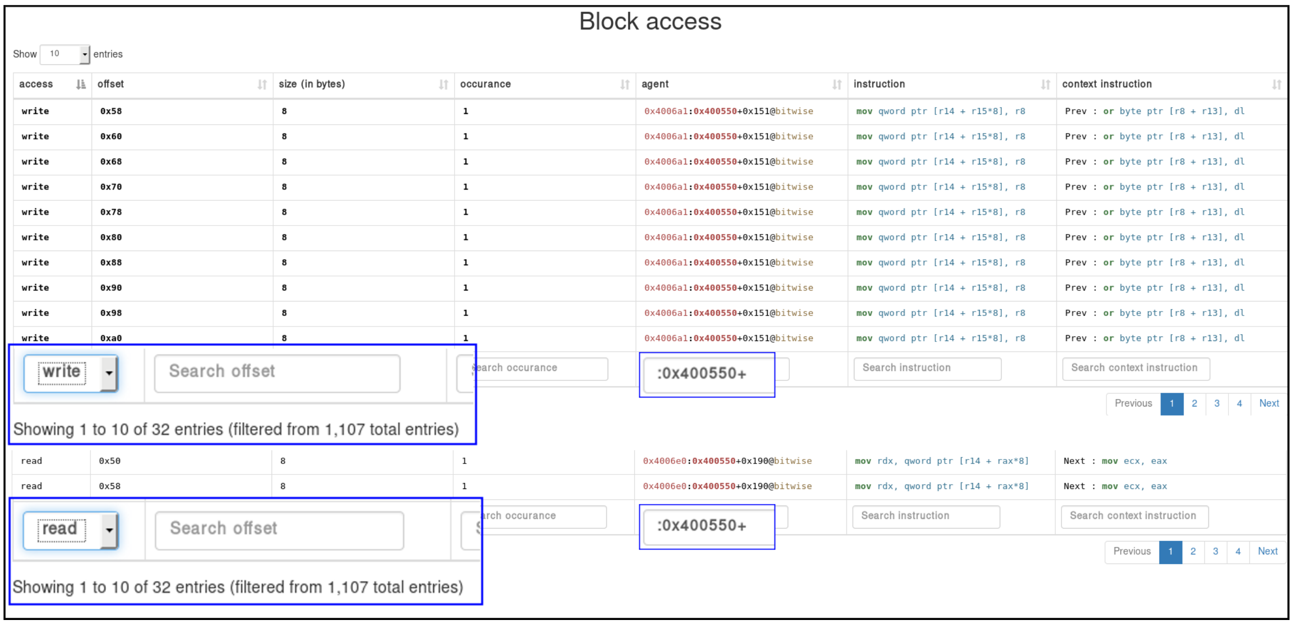
Detect instances

Show 10 entries

start	size	malloc caller	free caller	detailed view
0x604010	256	0x400593:0x400550+0x43@bitwise	never free	block_33
Search adress	Search size	Search malloc caller	Search free caller	

Showing 1 to 1 of 1 entries

Figure A.5: struct_1 view, recovered from analysis of the longest block



Block access

Show 10 entries

access	offset	size (in bytes)	occurance	agent	instruction	context instruction
write	0x58	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x60	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x68	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x70	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x78	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x80	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x88	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x90	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0x98	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	0xa0	8	1	0x4006a1:0x400550+0x151@bitwise	mov qword ptr [r14 + r15*8], r8	Prev : or byte ptr [r8 + r13], dl
write	Search offset		Search occurance	:0x400550+	Search instruction	Search context instruction

Showing 1 to 10 of 32 entries (filtered from 1,107 total entries)

Previous 1 2 3 4 Next

read	0x50	8	1	0x4006e0:0x400550+0x190@bitwise	mov rdx, qword ptr [r14 + rax*8]	Next : mov ecx, eax
read	0x58	8	1	0x4006e0:0x400550+0x190@bitwise	mov rdx, qword ptr [r14 + rax*8]	Next : mov ecx, eax
read	Search offset		Search occurance	:0x400550+	Search instruction	Search context instruction

Showing 1 to 10 of 32 entries (filtered from 1,107 total entries)

Previous 1 2 3 4 Next

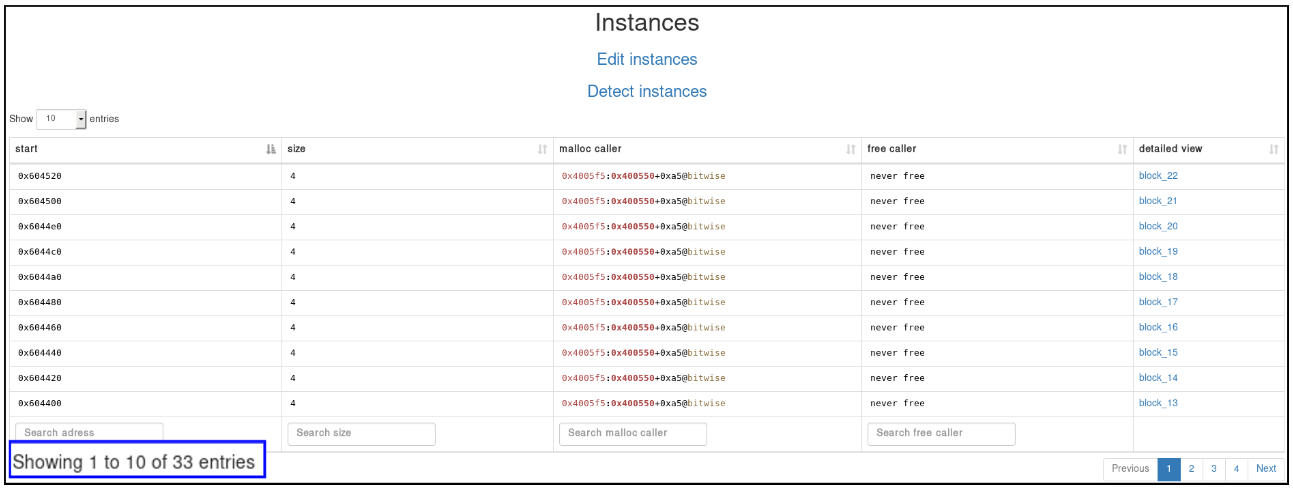
Figure A.6: Accesses made to the longest block in function 0x400550

maybe to check the result after some processing. The accesses made on this array by the allocating function are one read and one write per offset. This may mean the array is initialized in the allocating function and processing in the other function. I will call this other function the "processing function". To confirm that I will put the address of the other function in the search field of the column "agent". The result is 1043 of 1107 accesses are made by this function. So every access is made in the processing function except the 64 made in the allocating function.

Now I will analyze the detailed view of one of the little blocks. I to go the detailed view of block 22 and run the analysis on it. The "struct_2" is created and linked to this block. In the detailed view of this structure there are 4 members of the same type (int8_t) instead of 1 array with 4 units. This is due to the minimal size for an array which is 5. Next I click on the link "Detect instances" to automatically link every block which matches the pattern of struct_2 (in this case every array of four unit of int8_t). After that I can see there are 33 blocks in the second data-table of the page.This data-table displays the linked blocks or instances of this structure (see figure A.7). This means the 4-byte block which is not allocated at the same place is also an array of four int8_t.

Back in the detailed view of block 22, I will take a closer look into its accesses (displayed in the data-table). The first thing I notice is, there are only 9 write accesses (I checked that by selecting write in the bottom of the "write" column). All these write accesses are done in the allocating function. Four of these accesses are done by the same instruction (at 0x40068d) and are done to every offset. The instruction at 0x400628 also accessed every offset and an extra access is made at offset 3. All the instructions are quite close to each other. It could mean one or two loops are used to initialize the little blocks. Looking at the read accesses I notice some are done in the allocating function and some in the processing function. If I take a look at the offsets of the read accesses I notice that all offsets are read once in the allocating function and twice more for the offset 3. All of the other reads (24 in the processing function) access all the offsets between 5 and 7 times. Apparently these little blocks are allocated and then initialized in the allocating function, and after that, the processing function reads all the data they contain multiple times. Also their number is equal to the size of the bit representation of the input used when collecting the data ("4242"); this thought will be checked later in A.8 with the one relating to the size of the biggest allocation. I also check another of the little blocks and the accesses are very similar, so I consider all of them identical for the rest of the analysis.

Now it is time to take care of the lonely block, allocated at 0x4005a3 (block 32). In its detailed view I can see there are only 8 accesses, 4 writing and 4 reading. All of them occur eight times, and are done by the same instruction at 0x400705. This means this instruction does a read access and a write access at the same time, so this instruction is called 8 times per offset, which is the number of bits per offset. I will suppose this means the



Instances

Edit instances

Detect instances

Show 10 entries

start	size	malloc caller	free caller	detailed view
0x604520	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_22
0x604500	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_21
0x6044e0	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_20
0x6044c0	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_19
0x6044a0	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_18
0x604480	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_17
0x604460	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_16
0x604440	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_15
0x604420	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_14
0x604400	4	0x4005f5:0x400550+0xa5@bitwise	never free	block_13
Search adress	Search size	Search malloc caller	Search free caller	

Showing 1 to 10 of 33 entries

Previous 1 2 3 4 Next

Figure A.7: Auto-detected instances for struct_2

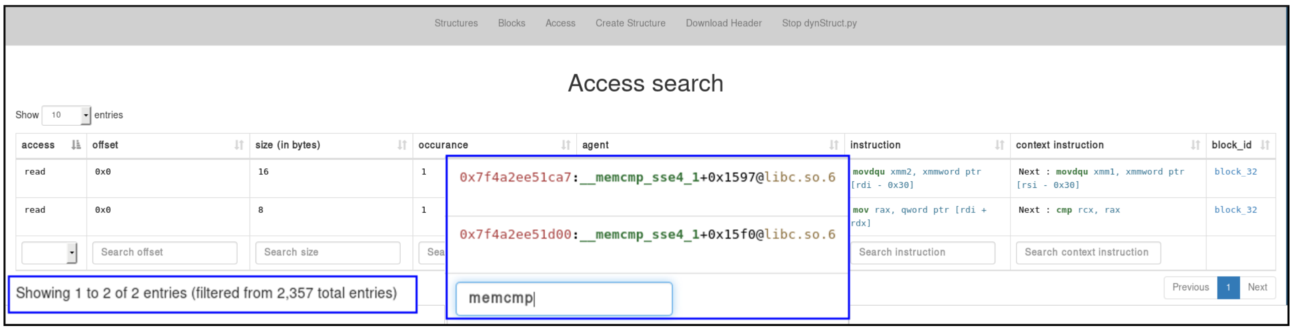
value store is built bit by bit. The fact that the instruction (it is always the same instruction which accesses the lonely block) which accesses the allocation is an "or" supports the idea of the bit by bit construction of the value.
Now we know that the allocating function also does the initialization of all the allocations. We also know that the processing function changes values of the array and reads every offset in the little blocks multiple times.

A.8 Step 5: Comparison of execution

Now I need to find out if my thoughts were correct or not to have a deeper understanding of `bitwise`. I had one thought which is that the size of the array and the number of little blocks is equal to the number of bits in the parameter given to the program. To check if this is true, I will re-execute `bitwise` under dynStruct with the argument "24" instead of "4242". If it is true I will have only 16 allocations and 16 units in the array.
The first command is similar to before: **"drrun -c dynStruct -o run_2 – ./bitwise 24"**. But this time for the script, it will be: **"python3 dynStruct.py -k -d run_2 -w -l 127.0.0.1:24243"**. The option "-o" in not used because I will use this only once to do the comparison, so I don't need to save the loaded data in a serialized file. The web interface is directly started via the "-w" option but on a different port (the "-l" option is used to change the port). Binding it on a different port allows me to have to two instances of the web interface in two tabs in the same browser to compare them easily. To finish the "-k" option is used because I know there are only arrays and not structures, so I have to tell dynStruct to not remove them (I don't have to analyze each of them via the web interface).
Looking at the details of the longest array, there are only 16 units, but there are no allocations of size 4. Instead I have 17 allocations of size 2, 16 by the same instruction and 1 separate. The one separate is allocated and used in the same way as the lonely one in the first execution; the 16 allocations are also very similar in their use (allocation, access pattern) as the little blocks in the first execution.
This confirms my thought. I also learn new information, the size of the allocations (little blocks and lonely block). This size is the number of bytes in the parameter given to `bitwise`.

A.9 Step 6: Gather and analyze libc memory use

In this last step, I want to know how the comparison with the hard-coded data is made and with what this data is compared.
If I take a look at the symbols imported by `bitwise`, I notice memcmp. Memcmp is used to compare memory areas; it could be used the check the input after the processing in `bitwise`.
To check that, I will do a third execution with the parameter "4242" but I will add monitoring of libc: **"drrun -c dynStruct -m libc -o run_3 – ./bitwise 4242"**. To start dynStruct.py I used the same command as the one used for the comparison (just change the name of the file).
This time I go to the access view to see if some accesses are done by memcmp. To check that I just write "memcmp" in the column agent as shown in figure A.8. I can see two accesses made in block



access	offset	size (in bytes)	occurance	agent	instruction	context instruction	block_id
read	0x0	16	1	0x7f4a2ee51ca7:__memcmp_sse4_1+0x1597@libc.so.6	movdqu xmm2, xmmword ptr [rdi - 0x30]	Next : movdqu xmm1, xmmword ptr [rsi - 0x30]	block_32
read	0x0	8	1	0x7f4a2ee51d00:__memcmp_sse4_1+0x15f0@libc.so.6	mov rax, qword ptr [rdi + rdx]	Next : cmp rcx, rax	block_32

Showing 1 to 2 of 2 entries (filtered from 2,357 total entries)

Figure A.8: All accesses made by memcmp during bitwise execution

32. If I go to the detailed view of this block it turns out to be the lonely block. This confirms that the lonely allocation is the one used to do a comparison, which certainly is the check to tell if the parameter is the flag or not. Some classical reverse engineering is needed to confirm that and finish solving the challenge.

A.10 Summary

By using dynStruct, I quickly know where to look (only two functions appear on the accesses) and what data structures are used by the program (I know the size of the different allocations). I also know where every allocation is done and accessed. All this information can be gathered faster using dynStruct than via classical reverse engineering.

A.11 End of solving

After this analysis via dynStruct, radare2 (a reverse engineering and exploitation framework [2]) was used to finish solving this challenge. It turns out the allocating function is the main of `bitwise`. After some reverse engineering it is possible to recognize a Burrows Wheeler transformation [23] based on the binary representation of our input (which explains the number of allocations and the size of the array). This explains the number of little blocks, which is "8 * <size of input>" or the number of bits in our input. After this transformation the Burrows Wheeler form of the input parameter is copied in the lonely allocation. This lonely allocation is compared with a hard coded array of 48 bytes after that (via the noticed memcmp). We don't have the key of the transformation but by reconstructing the matrix used for the Burrows Wheeler transformation using the hard coded data as last column of the matrix and only keeping the row starting with "PCTF{" (all flags in the plaid ctf start with these letters) it is possible to get the original sentence used. The Burrows Wheeler transformation is specially made to allow the reconstruction of the initial matrix. The final flag is "PCTF{clever_pun_about_burrows_wheeler_goes_here}".

A.12 Conclusion

All the thoughts I had during the dynStruct analysis were true. This information allowed me to only take a look at the important places in `bitwise`'s disassembly which were: the initialization (to know what are the data put in the allocations), the processing (to know what manipulation is done to the array) and the final check (to know what data are put in the lonely allocation and with what it is compared).

References

[1] Hex-Rays SA. IDA PRO disassembler and debugger. July 2016. https://www.hex-rays.com/products/ida/

[2] Radare2. July 2016. http://radare.org/r/

[3] The GNU Project Debugger. July 2016. https://www.gnu.org/software/gdb/

[4] X64dbg. July 2016. http://x64dbg.com/

[5] Daniel Mercier. dynStruct. August 2016. https://github.com/ampotos/dynStruct

[6] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. 2008. Digging for data structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (OSDI'08). USENIX Association, Berkeley, CA, USA, 255-266.

[7] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium* (CERIAS '10). CERIAS - Purdue University, West Lafayette, IN, Article 5, 1 pages.

[8] Chi-Keung Luk, Rovert Cohn, Robert Muth, Harish Patil, Arthur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'05), pages 190–200, Chicago, IL, USA, 2005.

[9] Asia Slowinska, Traian Stancescu, Herbert Bos. Howard: a Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of Network and Distributed System Security Symposium. The Internet Society, 2011.*

[10] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, 2005.*

[11] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of Network and Distributed System Security Symposium. The Internet Society, 2011.*

[12] Ed Robbins, Andy King, and Tom Schrijvers. 2016. From MinX to MinC: semantics-driven decompilation of recursive datatypes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 191-203.

[13] Michal Zalewski (aka lcamtuf). American fuzzy lop. July 2016. http://lcamtuf.coredump.cx/afl/

[14] Nicolas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07).* pages 89–100, June 2007.

[15] Stefan Le Berre. Vulnerability research on Windows binaries. In *Nuit du hack 2015.* July 2016. https://www.youtube.com/watch?v=IexI5hIY6A0

[16] Cosmin Gorgovan. dynamorio_pin_escape. July 2016. https://github.com/lgeek/dynamorio_pin_escape

[17] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12).* ACM, New York, NY, USA, 133-144.

[18] Google Inc. Google Summer of Code. July 2016. https://developers.google.com/open-source/gsoc/

[19] Frida. July 2016. http://www.frida.re/

[20] Capstone engine. July 2016. http://www.capstone-engine.org/

[21] Bootstrap. August 2016. http://getbootstrap.com/

[22] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81).* IEEE Press, Piscataway, NJ, USA, 439-449.

[23] Michael Burrows and David John Wheeler. 1994. A Block-sorting Lossless Data Compression Algorithm. *Digital Systems Research Center*, Research Report 124.

[24] Mark Allen Weiss. Data Structures and Algorithm Analysis in C. Addison-Wesley, 1996.