VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Reverse Engineering of an Obfuscated Binary

Kaishuo Yang

Supervisors: David Pearce, Ian Welch

Submitted in partial fulfilment of the requirements for
Master of Science.

Abstract

Reverse engineering is an important process employed by both attackers seeking to gain entry to a system as well as the security engineers that protect it. While there are numerous tools developed for this purpose, they often can be tedious to use and rely on prior obtained domain knowledge. After examining a number of contemporary tools, we design and implement a de-noising tool that reduces the human effort needed to perform reverse engineering. The tool takes snapshots of a target program's memory as the user consistently interacts with it. By comparing changes across multiple sets of snapshots, consistent changes in memory that could be attributed to the user action are identified. We go on to demonstrate its use on three Windows applications: Minesweeper, Solitaire and Notepad++. Through assistance from the de-noising tool, we were able to discover information such as the location of mines and values of cards in these two games before they are revealed, and the data structure used for input to Notepad++.

Contents

Chapter 1

Introduction

1.1 Motivation

As modern software has become more sophisticated, the potential damage resulting from a hacker gaining unauthorised access has also increased. In response, developers create better security mechanisms while hackers find new ways around them, leading to a never ending game of cat and mouse. The use of reverse engineering is crucial for both the hacker trying to find weaknesses in program code as well as the developer seeking to understand how an attack is executed. This interesting dynamic can be easily seen in a variety of fields from malware analysis to cheating in video games.

1.1.1 History of Reverse Engineering

Reverse engineering is the process by which an engineered object is deconstructed to extract information from it regarding the object's design and architecture. The need to understand an object created by someone else is as old as engineering itself, with records of Romans reverse engineering ships in 300 BC [49]. In more recent years, reverse engineering has made its way to the software world where engineers dissemble, debug and otherwise dissect other peoples code for their own purposes. The first formal definition of reverse engineering came about in 1990[19] as analysing a subject system to:

- Identify the system's components and their interrelationships and
- Create representations of the system in another form or at a higher level of abstraction

What this means is that reverse engineers generally aim to analyse a target program, understand how each program component works and then potentially create their own version, incorporating at least some features or functionality from the original program.

1.2 The Problem

Reverse engineering is the process of analysing a piece of software and understanding how its components work together in the full program. This is important because while most software only display their output to users, advanced users such as security researchers need to understand how internal components work. Because the reverse engineer does not have access to the source code, they must instead obtain domain knowledge from analysis of the compiled program. This can often be a difficult and tedious process as manual testing is often the primary method by which specific domain knowledge is obtained. Successfully reverse engineering software requires analysis of information flow within the program, commonly done either dynamically while the target program runs, or through static analysis methods involving disassembly or decompilation of the program executable.

Dynamic analysis involves analysing the program while it is running. This is often achieved by providing the target program with a controlled set of input data and then observing how this data is used and possibly transformed by the program. This must be repeated many times with different data sets in order to achieve a greater understanding of how the target program works. Both creating the test cases and running them are labour intensive as test frameworks, often designed for use with source code, cannot be used with compiled programs

Static analysis involves disassembling the target program and then analysing the assembly code produced to determine how it works. Even with modern disassembly tools it is still difficult to fully understand a program's function through disassembly because information is lost during the compilation process as source code is turned into machine code [25]. Important details that often cannot be retrieved from disassembly include:

- Function and variable names
- Comments explaining what confusing sections of code do
- Separation of code and data segments in the executable

All of the above need to be reproduced through human intuition and guesswork and as a result static analysis can be frustrating and tedious for the human engineer.

1.3 Contributions

This thesis makes contributions towards the creation of a noise reduction system used in reverse engineering. Through a literature review we have identified several shortcomings in current reverse engineering tools. Based on this analysis, we then design and implement a prototype tool that meets these gaps. Finally, we evaluated the effectiveness of our tool for reverse engineering several common Windows applications. Specifically, the main contributions of this thesis are:

- Literature review of reverse engineering methods and tools. After analysing a variety of contemporary tools, we identified several shortcomings that caused inefficiency when used to reverse engineer programs:
  - Difficult to locate important data in program memory as it is tedious to identify changes in memory resulting from user actions. This is because current dynamic analysis methods often cannot distinguish between changes caused by a user action as opposed to those caused by other sources.
  - Many of the current tools require prior domain knowledge to successfully perform reverse engineering on the target program. Without an idea of the target's data layout or function structure, the output from advanced tools such as IDA are hard to understand.
- Development of a new tool that meets these shortcomings. Having identified problems with existing tools that complicate the reverse engineering process, we then set out to design and implement a prototype tool to meet these gaps. The key features of our tool are:

  - Usage of intersecting snapshots to enable accurate analysis of program memory to attribute changes in memory to user actions. The tool first obtained sets of memory snapshots before and after a user action is performed. By using a process of intersecting sets, locations that changed consistently are isolated, allowing the user to accurately see the changes
  - Analysis based purely on user actions without need for the user to have prior knowledge of the system. As it is designed to be snapshot program data while a user interacts with the program normally, it avoids the domain knowledge requirement set by other tools
  - A storage system for memory snapshots that saves snapshot sets with the action performed to achieve them. This allows future analysis of any experiment data without loss of information.

- Evaluation of the tool on Windows applications to assess its effectiveness. We tested the tool on three applications: Minesweeper, Notepad++ and Solitaire to see if it could assist in reverse engineering. Through testing we found that the system was a useful prototype that successfully removed noise to simplify the reverse engineering process. However certain shortcomings were also identified which leads to potential to improve the tool.

1.4 Thesis Organisation

The thesis is organised as follows:

- Chapter 2 presents background information about the uses of reverse engineering in fields such as malware analysis and program disassembly. It also examines current tools and challenges of performing reverse engineering
- Chapter 3 covers the iterative design process of our denoising tool
- Chapter 4 demonstrates the use of our denoising tool on a number of case studies to illustrate how a typical reverse engineer would make use of this tool to achieve their goals
- Chapter 5 concludes the thesis with some final thoughts.

Chapter 2

Background and Related Work

2.1 Malware analysis

Malware is a broad term referring to any form of ″malicious software″, i.e. software designed to damage or allow unauthorised access to a user′s computer [35]. Malware is always created with malicious intent and includes common forms such as viruses, Trojans, and worms. Such threats can only activate after being introduced into the user′s machine through some attack vector.

With malware becoming an ever prevalent problem [32], security researchers, such as anti-virus developers, must find new ways to stop both the introduction and activation of malware in order to minimise potential damage done by them to client computers. In order to develop effective countermeasures, researchers must first figure out how malware works. Since current, active malware source code is not usually released by their creators in order to hide how their programs work, researchers must reverse engineer malware executables in order to discover their inner workings [16]. Malware analysis can be done either statically or dynamically [64].

2.1.1 Static malware analysis

Static malware analysis is performed by examining the program′s code without executing the actual malware in its entirety. Often this involves disassembling the malware then examining each component and studying its inner workings to gain a better understanding of the malware as a whole. By examining the disassembled code, researchers can gain some insight as to how parts of the malware work and even how these components interact to cause harm. This is often a safe and controlled way to analyse malware as it does not get a chance to cause damage to the user′s system. Static malware analysis is also performed by anti-virus software when they scan files for malware. This form of static analysis looks for certain characteristics, known as signatures [39], of malware such as a common byte sequence and uses the presence of these signatures to detect the malware. For example, a malware that installs unwanted ads on a victims computer (known as ″adware″) will likely link to its creator′s website. If we identify the creator′s website URL, we can say with some confidence that any program linking to this URL is a variant of the adware. In this case, the website URL can be considered the malware′s signature.

In order to hide their creations from antivirus systems, malware authors employ a variety of techniques to obfuscate malware and reduce the chances of detection [45]. The most common form of malware obfuscation involves compressing the malware executable resulting in a form of archived file[52]. Known as ″compressors″ , this enables malware to be decompressed into memory and then executed. This process makes the detection of malware difficult as antivirus programs must first decompress the binary without accidentally executing it before analysis is possible. Other forms of malware obfuscation include ”crypters” that simply encrypt malware without compression, making it difficult for antivirus programs to decrypt and analyse the program. Finally ”protectors”, more commonly known as ”packers”[67] combine features from both compressors and crypters to create a encrypted and compressed program that is difficult for other programs, such as antivirus, to fully analyse. On the other end, ”unpacking” refers to the process by which a packed program is decompressed and decrypted to allow effective analysis. As malware has become more sophisticated [15], new methods to decrypt and statically analyse these programs have also appeared to counter the rising threat [37]. A number of these examples are discussed below.

Eureka is a framework designed to assist with static analysis of packed malware by deobfuscating the target program and executing it in a controlled virtual environment [59]. During execution, Eureka uses a special kernal driver to detect programs unpacking and executing its malicious payload. The detection is done by detecting system calls used by common packing programs. For example, if a program begins to Once the malware has unpacked its payload, Eureka dumps the process to disk and uses IDA Pro to disassemble and analyse the code. This provides a visual representation of the program flow, enabling specialists to better analyse and understand the malware.

AppSpear, an automatic unpacking system designed to unpack obfuscated Android applications [68] was developed in response to the rise of malware targeting mobile devices. It decrypts application at the bytecode level and reassembles the bytecode, thus resulting in the creation of an unobfuscated binary. This binary can then be examined by program analysis tools or automated malware detection systems. During testing, both the packed and unpacked binaries were tested for sensitive behaviours which could indicate the presence of malware. Out of 31 malware samples tested, there were a total of 208 sensitive behaviours detected in the packed binaries compared to 2493 in the unpacked versions. This system of unpacking a binary before analysis was thus shown as effective in correctly detecting malicious behaviour.

A novel way to study packed malware through the extraction and analysis of API calls was presented by Alazab, Venkatraman and Watters [14]. The group obtained a collection of malware, unpacked each sample to obtain the payload and then extracted API calls used from the dissembled binary. By extracting API function calls from known malware, a database of suspicious API calls often used by malware was assembled, which was then sorted into six main categories: file searching, file copy and deletion, obtaining file information, moving files, reading and writing files, changing file attributes. These details can then be used as signatures for classifying future programs as malware or normal software.

2.1.2 Dynamic malware analysis

Dynamic analysis is performed by observing how a piece of malware behaves at runtime [56]. This is often performed on malware running inside a controlled sandbox environment in order to obtain accurate information on the malware without any risks of it causing harm to a real system. By using a sandbox environment, often a virtual system functionally identical to a real one, any damage can be easily analysed then repaired by rolling back the sandbox to an initial state, allowing not just safe, but also quick and efficient analysis. At runtime a debugger can be attached to the malware to monitor its actions in real time. Unfortunately, as malware become more sophisticated, they have been able to detect the presence of attached debuggers or become aware that they are being executed within a sandbox environment. When either of these conditions are detected, the malware usually self terminates to hinder any efforts at reverse engineering it. As a counter to such behaviour, advances in reverse engineering led to strategies being developed to discreetly monitor programs in a sandbox environment. A possible strategy is dynamic instrumentation which refers to programs on the sandbox's host machine being able to directly monitor the behaviour of programs within the sandbox (guest). Dynamic instrumentation is discussed in further detail in Section 2.2.

However, not all dynamic analysis is done in a virtual environment [62]. For example, anti-virus programs must run in realtime on the users computer, analysing and preventing suspected malware activity. Malware authors often use methods to obfuscate their code so that the malware is not detected by traditional syntactic anti-virus systems. This is problematic as specialists need to wait for each new executable or variation of the malware to attack a number of machines in order to identify a file signature. As a result, it is beneficial to examine a program's behaviour rather than code signature to determine whether it is malware or not. A number of ways to dynamically analyse obfuscated malware have been devised, such as:

- Investigating differences in opcodes used by the malware compared to those used by benign software
- Stealthily debugging malware by breaking its code stream into chunks and debugging each chunk
- Analysing programs to determine what system resources are being used

One way of identifying malware presence in real time is the identification and detection of system opcodes likely to be used by such programs [55]. This is similar in concept to the Warden program discussed in section 2.5.5 using the presence of certain assembly instructions (JMP, INT3) to detect unauthorised programs accessing a game's memory region. By looking at the frequency different opcode sequences appear in both a collection of malware and benign programs, we can calculate how likely it is for a given opcode sequence to appear in a malicious program. During experimental testing done by researchers at the University of Deusto, opcode sequences of length 1 and 2 were used to compare identified malware with both their variants as well as benign software. It was found 60% of malware variants exhibited 90% similarity in these opcode sequences compared to their original versions. On the other hand, 85% of benign software exhibited 60% or less similarity in opcode sequences when compared to the original malware. This shows that it is possible to identify malware at runtime based on the opcodes they use.

Cobra is another framework designed to enable stealthy dynamic analysis of programs and identify malware as they are executed [63]. It implements a concept of "stealth localised execution" where a program's code stream is broken up into several groups (blocks) of instructions at runtime. Specific code constructs, such as debugging commands, can be added to each block in order to allow analysis of the code. These blocks are then executed in order, one at a time, to mimic normal program execution while enabling program analysis without alerting the target program of the presence of the debugger. As a result, malware can be successfully studied without triggering their self destruct functionality.

Another common tool used for dynamic analysis is the Microsoft Process Explorer tool which monitors actions performed by active programs in real time [54]. It identifies resources that are being used by a process in order to track down problems with a program or identify a program as malware if it is accessing unexpected resources. This tool uses Windows kernel-mode functions and is able to divulge information not normally available to userland debuggers, such as which threads are using the CPU.

2.2 Dynamic instrumentation

Dynamic instrumentation refers to the ability to monitor a program's behaviour and performance while it is running [51]. Tools used for dynamic instrumentation usually aim to support important features such as step-by-step program execution, debugging [70], performance analysis [38], and data and event logging [47]. These tools are often used by security researchers to monitor malware execution in controlled environments and understand its mechanisms in order to develop anti-malware tools.

In order to hide their presence, many dynamic instrumentation tools attempt to be invisible to the application being debugged, i.e. applications being monitored can only "see" operating system components such as main memory and cache without being able to detect any activity associated with dynamic instrumentation [51].

DynamoRIO [18] is a tool used in the industry for dynamic instrumentation of applications. It virtualises processes by copying the application's code into a cache then executing instructions from the cache along with additional instructions generated for dynamic instrumentation. Each memory address that is accessed by the application is either an original application address, or an instruction cache address that has been translated to an original application address. Furthermore, any indirect branch targets are automatically converted from application addresses to addresses in the instruction cache. This results in the dynamic instrumentation becoming very transparent, and makes it difficult for the application to detect that it is being monitored.

MALT is a newer approach used to debug stealthy malware that operates at system-level ("ring 0") which can evade most virtualised and bare-metal debugging attempts [69]. Instead of virtualising the malware as DynamoRIO does, MALT executes at the System Management Mode (SMM also known as "ring -2"). SMM is a mode present on x86 CPUs designed to be used by firmware to control very low level features such as power or system hardware functions. Executing a debugger at SMM level grants it high privileges, including being able to suspend even the computer's operating system. As a result, MALT is able to debug the malware without displaying artefacts that alert the malware of its presence. Testing of MALT showed that is it invisible to many popular anti-debugging and anti-virtualisation strategies while it is also able to implement common debugging tools such as breakpoints and step-by-step program execution.

ANUBIS [61] is a framework of several tools for dynamic code analysis of unknown programs. It was created to automate analysis of malware as there were too many samples to be completely analysed by humans. It works by first running the target unknown binary in an emulated PC environment and then monitoring its behaviour. Because it is done in a controlled virtual machine, it is difficult for the malware to detect the presence of a monitoring tool that has full control over its virtual operating environment. Actions such as system calls and Windows API functions are tracked and then a detailed report is generated. Based on this report, experts can decide whether or not the target binary requires further detailed analysis.

2.3 Buffer Overflow

A buffer overflow occurs when a program writes more data to a buffer than it can contain, resulting in memory locations beyond the buffer becoming overwritten. Buffer overflow attacks are often performed by malicious software where by overflowing the buffer, malware can write to memory locations it does not normally have permission to access. A properly executed buffer overflow attack can, for example, allow the attacker to modify the return address of functions within a program, and let the malware re-route program flow and execute

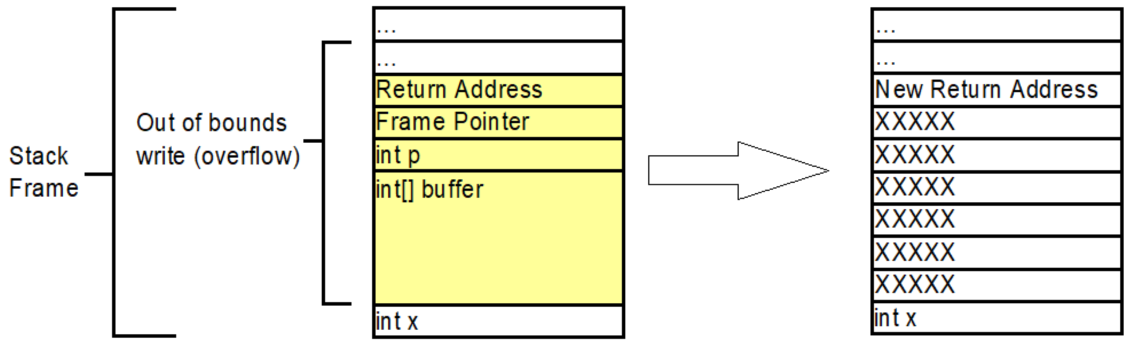


Figure 2.1: Visualised stack frame during a buffer overflow attack. By writing to the `int[] buffer` beyond its capacity, an attacker can overwrite the return address to alter the intended program flow. During this process, both `int p` and the frame pointer are also overwritten. `XXXXX` represents junk data bytes inserted into the `int[] buffer` in order to get to the return address.

commands at the same privilege level of the targeted, vulnerable program [36]. As a result, detecting buffer overflow is important as it is an example of detecting possible intrusions made by an attacker. For example, consider the code fragment below:

```
f(int p){
        int[] buffer
        int x
        ...
```

During execution as shown in Fig 2.1, an attacker has overwritten the function's return address by overflowing the *int[] buffer*. This changes the intended program flow and it is often difficult to detect the presence of such an attack.

Due to the risks associated with such attacks, detecting and preventing buffer overflow vulnerabilities is a commonly studied topic in software security. A wide range of defensive mechanisms have been developed, such as runtime analysis to detect invalid malloc and free calls, static analysis to determine the safety of each pointer used [71], and finally the use of dummy values to detect unwanted memory overwriting. The use of dummy (canary) values to detect overflow, implemented in the "StackGuard" patch for the gcc compiler [2], has also been used commercially by Blizzard to detect memory access in World of Warcraft. When a program is compiled, StackGuard adds an otherwise useless 'canary' memory block before a function's return address. Consider the code fragment above, but this time StackGuard has been used during compilation which implants canary values in the program memory space and maintains a record of these canary values elsewhere. An attacker attempts a buffer overflow attack as shown in Fig 2.2, again overflowing the *int[] buffer* in order to corrupt the return address. However this time, they have to overwrite the canary value before the return address. The program is able to detect a change in the canary value thanks and self terminates to prevent the attack.

In games such as "World of Warcraft", the game security system uses a more advanced type of canary value known commonly as "trapped memory" segments. These regions of memory appear to contain relevant game information, such as player location data, but are actually just non functional and often unencrypted copies of the real data. Any attempts to modify these "trapped" regions of memory results in the detection of unauthorised access due to the mismatch between the data stored in the "trapped" region and the memory ad-

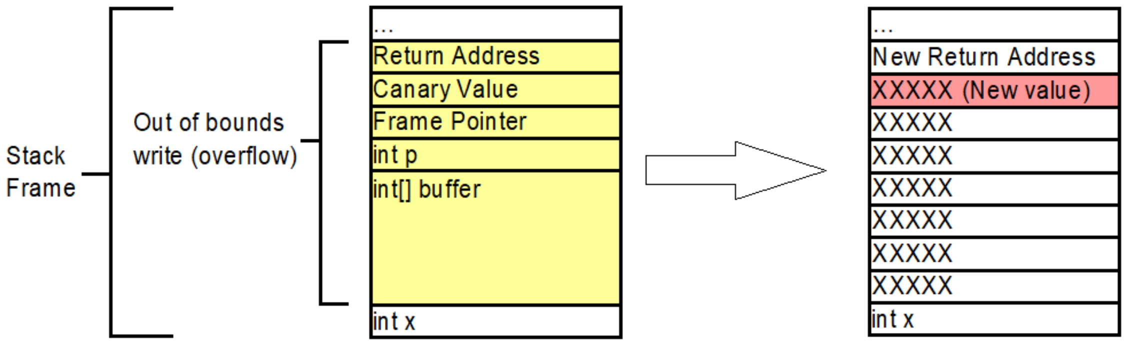


Figure 2.2: Visualised stack frame during a buffer overflow attack with StackGuard protection. The attacker has, just like in Figure 2.1, attempted to attack the program by overflowing `int[] buffer` in order to alter the return address. However this time, they have also overwritten the "canary value" provided by StackGuard. As the program maintains a record of expected canary values and locations, it notices that this canary value has changed and thus self terminates to stop the attack.

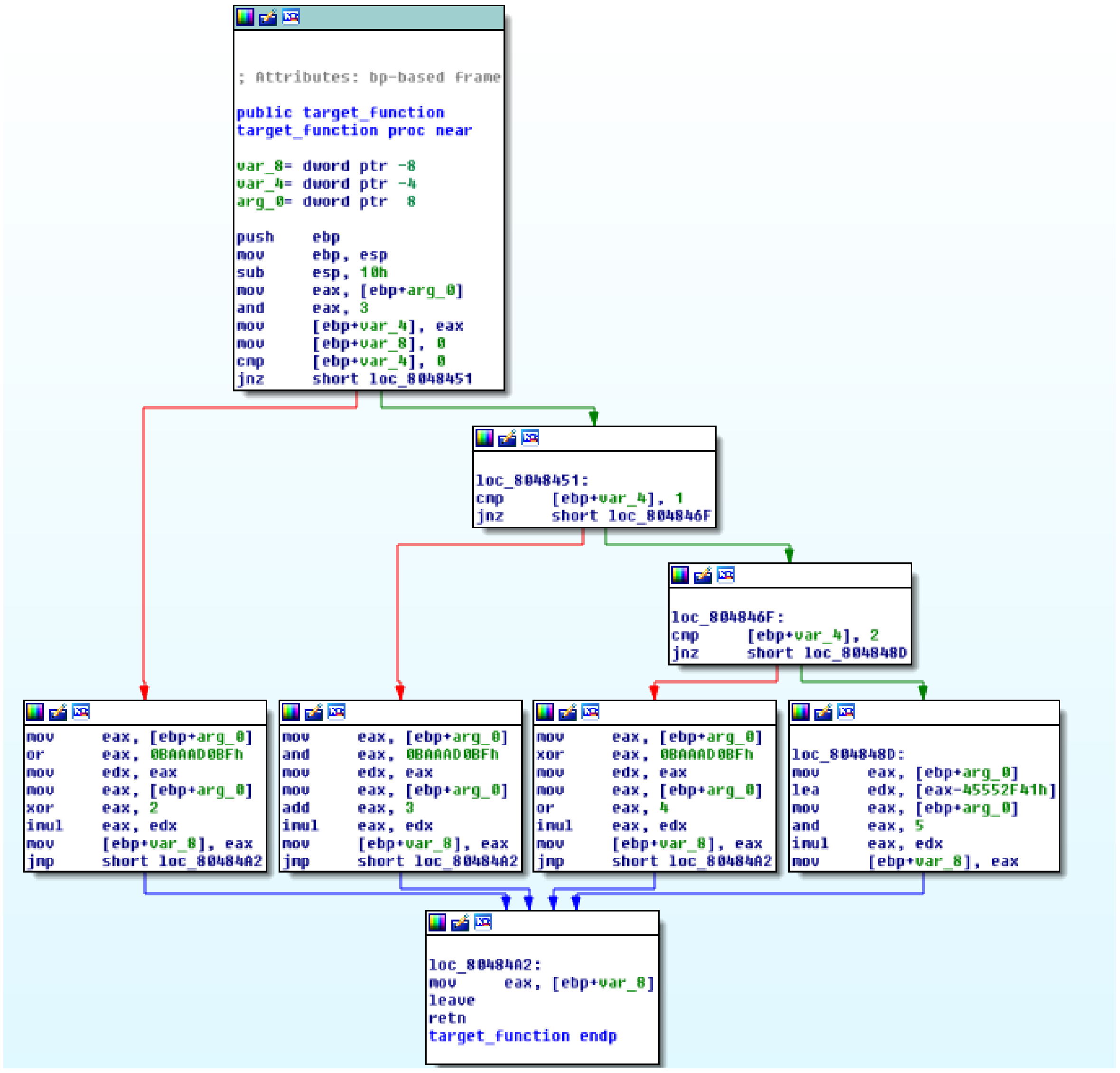
dresses containing the real game data. Attempts in evading such buffer overflow detection techniques have included identifying and saving canary values prior to an attack, and reinstating them after the return address has been overwritten. However, due to the recent heavy obfuscation of the such security systems, identifying trapped memory is difficult and as a result similar strategies used in the past to counter this protection are no longer effective.

In addition to the approach of using canary values to detect buffer overflow, there have also been ways to protect against overflow attacks without modifying the source code. Both CCured [44] and a later development, Softbound [43], aim to detect overflow attacks that modify the expected pointer values of applications. At compile time, pointer base and bound metadata are stored in a separate storage system and runtime checks inserted into code. These checks cause the storage to be checked upon load and store of pointer values to ensure there has been no unexpected modification. During testing all forms of spatial violations were detected showing that this system of checking pointer values is an effective defence against buffer overflow attacks. However, the overhead from using such a system was also significant at up to 127% with Softbound and 83% with CCured.

2.4 Reverse Engineering

Disassemblers work by parsing executable machine code into assembly instructions. Some programs are able to be disassembled by reading the executable machine code until a valid assembly opcode is encountered and then decoding the next few bytes, depending on the expected size of the decoded instruction [58]. This process is then generally repeated for the entire program, producing an accurate assembly representation due to the one-to-one relationship between assembly and machine code [66]. However, this approach may run into difficulties when data and code become mixed together in a program. It is possible for data to reside in the code portion of a program due to constructs such as constant strings, and likewise possible for code to reside in the data portion of a program. As a result, a disassembler that simply parses machine code will be unable to distinguish between the two.

Due to these limitations, modern disassemblers such as IDA Pro are more interactive



```
; Attributes: bp-based frame

public target_function
target_function proc near

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 10h
mov     eax, [ebp+arg_0]
and     eax, 3
mov     [ebp+var_4], eax
mov     [ebp+var_8], 0
cmp     [ebp+var_4], 0
jnz     short loc_8048451
```

```
loc_8048451:
cmp     [ebp+var_4], 1
jnz     short loc_804846F
```

```
loc_804846F:
cmp     [ebp+var_4], 2
jnz     short loc_804848D
```

```
mov     eax, [ebp+arg_0]
or      eax, 0BAAAD0BFh
mov     edx, eax
mov     eax, [ebp+arg_0]
xor     eax, 2
imul    eax, edx
mov     [ebp+var_8], eax
jmp     short loc_80484A2
```

```
mov     eax, [ebp+arg_0]
and     eax, 0BAAAD0BFh
mov     edx, eax
mov     eax, [ebp+arg_0]
add     eax, 3
imul    eax, edx
mov     [ebp+var_8], eax
jmp     short loc_80484A2
```

```
mov     eax, [ebp+arg_0]
xor     eax, 0BAAAD0BFh
mov     edx, eax
mov     eax, [ebp+arg_0]
or      eax, 4
imul    eax, edx
mov     [ebp+var_8], eax
jmp     short loc_80484A2
```

```
loc_804848D:
mov     eax, [ebp+arg_0]
lea     edx, [eax-45552F41h]
mov     eax, [ebp+arg_0]
and     eax, 5
imul    eax, edx
mov     [ebp+var_8], eax
```

```
loc_80484A2:
mov     eax, [ebp+var_8]
leave
retn
target_function endp
```

Figure 2.3: IDA Pro Control Flow Graph resulting from dissembling a function containing multiple if statements [50]

in the disassembly process [60]. Interactive disassemblers, as their name suggests, actively interact with the machine code while disassembling it. This means that they make use of dynamic analysis tools such as debuggers alongside the traditional disassembly methods to actively connect related code segments within the program [3], resulting in a graphical view of the code's structure. A control-flow graph, such as the one shown in Figure 2.3, may be generated during this process can be used to aid in further understanding of the program's components.

On the other hand, a decompiler seeks to generate high level (such as C or Java) code based on a program's binary or assembly code. Even for a single language, there are many different versions of programs that could result in the same, or very similar, assembly code. For example, *switch*, nested *if* and multiple *jump* statements could all be functionally identical and result in a series of *JMP* assembly instructions. As a result, a decompiler can only make an educated guess based on the control-flow analysis of the program when it is producing output and any code produced will only be a possible version of the program without any guarantees of resemblance to the real source code. Furthermore, obfuscation such as the stripping of symbols means that it is difficult to identify variables and function names as well as establish meaningful connections between them [26].

Ollydbg is a x86 debugger often used for reverse engineering of programs [56] with a built-in disassembly and assembly system. It is popular amongst both engineers seeking to verify correct operation of their own software as well as hackers looking to crack programs made by other developers. Ollydbg traces most program actions such as system and API calls as well as provides a user friendly interface for working with program memory. By providing these features, a reverse engineer is able to disassemble a piece of software, modify it then reassemble it.

Another approach to reverse engineering includes the use of binary analysis tools to examine code at a binary level. By investigating features such as data types and control paths in a binary, it is possible to gain insight into the otherwise hard to analyse workings of a program without having access to its source code [41]. This is not an easy task as a variety of factors make binary code complex and difficult to understand. These include both incidental causes such as compiler settings and runtime libraries, as well as deliberate measures taken to obfuscate binary code. For example, a tool called Zipr [29] rewrites binaries efficiently to create an obfuscated binary whose CPU and memory footprint is less than 5% more than the original [31]. This results in a program that is functionally identical to the original, but has vastly different binary code, making it harder to perform binary analysis. As a result, a number of innovative ways to perform binary analysis have been created.

Examples of newer binary analysis tools able to defeat obfuscation include BINSEC/SE and BinGold. BINSEC/SE [21] is a new dynamic symbolic execution engine that enables modular analysis of target binaries. It is able to perform functions such as obtain function parameters and return values from a binary and worked well in analysing and reverse engineering malware as well as deobfuscate binaries. While different in implementation, BinGold [30] is another binary analysis tool that defeats obfuscation by examining both data and control flow to find similarities between binaries. Because data and control flow are generally unaffected by light obfuscation or refactoring, BinGold performs better at tasks such as identifying program authorship and presence of copied components.

While most binary analysis tools are platform dependant, there has also been attempts to develop tools to support multi-platform analysis. For example, REV.NG [27] is a binary analysis framework based on QEMU that allows the user to recover CFGs and function boundaries from binaries built for a variety of system architectures. QEMU is a machine emulator that uses dynamic binary translation to parse binary code from any of the supported architectures into a custom intermediate representation (IR). By performing binary analysis on this custom IR, REV.NG was able to extract CFGs and function boundaries from a number of different binaries.

As opposed to most previous tools focusing on optimising the back-end analysis, B2R2 [34] is a binary analysis framework focusing on an efficient front-end design. The front-end consists of the disassembler and lifter which had been ignored by researchers in the past who optimised analysing the intermediate representation that is produced by the front-end. By using strategies such as parallelism in its analysis, B2R2 was able to perform binary analysis with an order of magnitude improvement in efficiency.

2.5 Reverse Engineering in games

A key, yet lesser known, field where reverse engineering plays a pivotal role is in the development of unauthorised game modifications and the security measures used to counter them. Common examples of these modifications include

- ″Cheats″ that enable players to gain an unfair advantage over others by gaining access to additional information or actions not normally possible in games
- ″Bots″ that automate normal gameplay for the player

We will examine the development of such modifications in the game World of Warcraft to exemplify the reverse engineering tools and techniques used.

2.5.1 World of Warcraft game

World of Warcraft (WoW) is a massive multiplayer online role-playing game (MMORPG) with a subscription pay model developed by Blizzard Entertainment. Each player controls a virtual character with which they participate in a range of activities such as combat against computer controlled monsters or other players, gathering of materials and crafting items. Players can also interact with each other cooperatively and team up to tackle difficult objectives or engage in trade. The prospect of a virtual reward encourages players to spend time doing repetitive and boring tasks (commonly known as grinding). For example, a player may spend many hours in-game catching fish to sell for in-game currency (gold), which is desirable as it is used to buy in-game items, and can be converted to real world money. As a result, the pride and accomplishment that comes from the gold earned compels some players to engage in this type of boring gameplay, commonly known as ″gold farming″, for long periods of time.

In contrary to boring tasks such as gathering of ores (mining), there are many activities players consider fun and interactive, such as combat against challenging enemies. Players wish to do these fun tasks themselves, but in order to access such activities, players must often first spend time doing the more boring tasks. For example, a player may need to spend many hours mining ore in order to craft a sword that they can then use to improve their combat capability and make them able to defeat more powerful monsters. This is an intended part of the gameplay loop designed to artificially prolong content by increasing the time needed for players to complete the fun activities. By increasing the time taken for players to complete all activities, players are kept subscribing for a longer period of time which results in more revenue for the developers.

As a result, programs, known as ′bots′, were designed with the goal of automatically performing such tasks. These bots could play the boring parts of the game, leaving the human player to better spend his time on the more interesting parts. Because bots bypass the basic gameplay loop of ″spend time to get rewards″ implemented by the developers, they are perceived by some as cheating [17]. In order to preserve the game′s integrity, Blizzard has developed a sophisticated tool, Warden, to detect and ban players using bots. Blizzard have been largely successful, with over one hundred accounts detected and banned every few months [10] [11] [12]. Facing potentially a million accounts lost and unable to evade Blizzard′s new technology, large scale bot developers have largely ceased operations [8] [1].

2.5.2 Game automation programs

Game automation programs (also known as ′bots′) are often used by players to complete boring tasks for them. Two such examples include *gathering bots* and *combat bots*. Gathering bots primarily focus on the gathering of raw resources such as fish, herbs and ore. On the other hand, combat bots focus on fighting computer controlled monsters to gather experience and items.

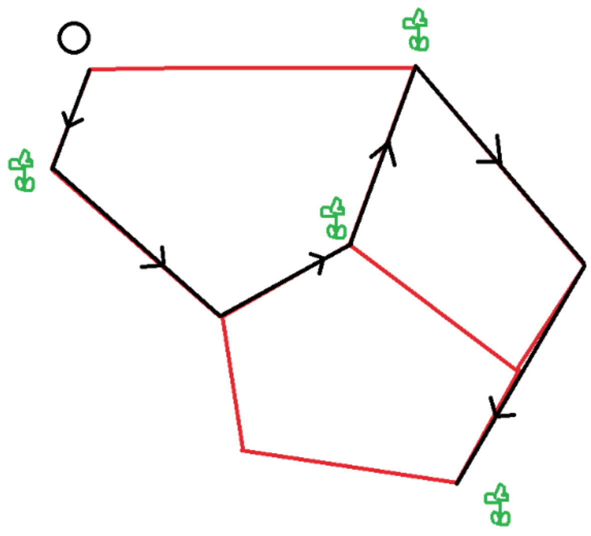


Figure 2.4: An example of navigation mesh based pathfinding. The black path is one that a bot (circle) is likely to take within the navigation mesh(red) when collecting from 4 herb nodes (green) available.

Gathering bots

Gathering bots gather resources from nodes such as herb bushes or ore veins in the game world. Because each node disappears after being harvested from and is re-generated in a different location, gathering bots must make use of a pathfinding system to travel between nodes. Nodes may appear randomly in a number of fixed locations and as a result a useful navigation mesh will encompass all known node locations. Bots can then use this navigation mesh to travel from one node to the next, as shown in Figure 2.4.

When a bot reaches the target, it must interact with the node to harvest resources. This interaction is the equivalent of a human player performing a single click on the node, at which point the resources within are deposited into the player's inventory and the node disappears. After harvesting a node, the bot will then move to the next nearby node and repeat the process.

Combat bots

Combat bots traverse the game world and kill enemies that they encounter which is desirable as players are rewarded with items and experience points for defeating enemies. Because monsters disappear when killed and then reappear in a different location after a short time, combat bots have to be capable of movement.

Combat bots can operate on a navigation mesh system similar to gathering bots, but some also function using direct movement between objectives. Bots using direct movement often rely on automatic use of in-game functions such as "target creature with specified name" and "move to target" which are provided by the developers for legal use by human players through ingame actions. As a result, bots using these functions have more similar movement characteristics to humans compared to bots using a navigation mesh system.

For example, consider the scenario in Fig 2.5 where a player character (black circle) must kill the four monsters (black pigs). There is a distinct difference in pathfinding behaviour between a navigation mesh based bot (red path) and a direct movement bot (green path). This is detectable using techniques such as the lowest common prefix algorithm [42] to dis-

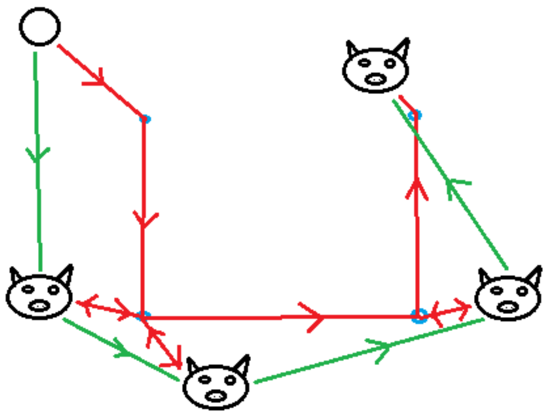


Figure 2.5: A comparison in pathfinding behaviour between combat bots using navigation mesh (red) and direct movement (green)

tinguish between human and waypoint based bots and will be discussed in detail in Section 2.5.5.

2.5.3 Development challenges

The first challenge in developing a functional bot is getting the program to obtain information from, and send commands to the game. After the bot is able to communicate with the game, it must also deploy defensive mechanisms to evade the game's built-in bot detection. Failure to do so will result in the user's account being suspended and render the bot useless.

Most bots obtain game state information and issue commands by reading from and writing to the game's memory region. By directly accessing game variables, intrusive bots have access to a very accurate representation of the game state in real time, and are able to make fast and accurate actions. Development of an intrusive bot requires extensive knowledge of the game's architecture and memory space, which change with every new game version, leading to high development costs. However, because this structure is standardised across all copies of the game and intrusive bots are easier to deploy and use, all commercial bots are of the intrusive type as they require connecting to the game's process before using [24] [23]. In recent years due to their prevalence, Blizzard has focused bot detection and prevention countermeasures towards such bots, which has significantly increased their risk and reduced their popularity.

2.5.4 Game developer and hacker interactions

An early prototype of bot detection software (Warden) was shipped with World of Warcraft's launch in 2004. It was initially crude and only functioned as a signature based detection system that scanned the user's computer for signs of known bot programs running, similar to an antivirus system. Over time however, Warden has been significantly upgraded with new technology to detect and prevent the use of bots. Understanding Warden's detection mechanisms is a key challenge for bot development as failure to do so will result in account suspensions, nullifying any advantages gained from using the bot.

Warden exists as a dynamically loaded module for World of Warcraft that is downloaded from the Blizzard servers and run several times a minute. After loading, it scans the game's allocated memory region and looks for external programs hooking functions or injecting dynamically linked libraries (DLLs), methods primarily used by bots to receive information from and send instructions to the game. Its memory scanning capability was once able to scan all memory addresses and processes on the user's computer but also lead to significant backlash and accusations of spyware [65] [40]. Furthermore, some programs have a legitimate need to monitor the game and while more popular ones such as Windows system drivers and Fraps (a screen recording software) are whitelisted, Warden has falsely flagged and banned others.

2.5.5 Later advancements

More recent reverse engineering of the World of Warcraft executable and Warden modules has revealed that Warden in fact scans a set number of memory locations known to be commonly accessed by bots. The data from the scans is then hashed and sent to the Blizzard server to compare against stored reference values and any irregularities are recorded. Abnormalities include JMP (jump unconditional) or INT3 (temporary replacement) instructions at the start of a function's memory space where such an instruction is not expected as the presence of such instructions indicate attempted instrumentation of the target program. This is indicative of an external application hooking a function into World of Warcraft, such as a bot writing mouse click coordinates to the game's memory. Detection of abnormal instructions in the game's memory space also served to deter earlier attempts at attaching debuggers to the game executable. Debuggers attach to the game process and control its execution via the use of breakpoints. These are created by saving the instruction at the breakpoint and replacing it with an INT3 instruction. When the CPU gets to the INT3, a software exception is generated and the CPU hangs, pausing program execution. When the debugger wishes to resume the program, the saved instruction is loaded back into its original location and program execution continues as normal. Because INT3 instructions are not present in the game during normal usage, any presence of such instructions is a clear sign of external tampering. This type of detection is similar to that used by malware to determine if it is being executed and debugged in a controlled virtual environment.

Another irregularity that Warden looks for are calls on the Lua call stack originating from outside of the WoW.exe memory region. World of Warcraft allows 3rd party user software written in Lua (a common scripting language), known as "addons", to offer additional functionality within the game client. Addons are normally only allowed to use a subset of lua functions available within the game, with many protected functions being reserved for use by the game client itself. Bots may need to escalate privileges and call protected system Lua functions through a process known as "Lua unlocking". Because the system Lua call came from an external program and not the WoW executable, Warden considers it evidence of tampering with the game client and the user account is flagged for potential punishment.

To counter Warden's scanning, large bot developers reverse engineered Warden's systems and discovered which memory locations were being scanned. As a result, bots avoided accessing these locations, and if a scanned location was accidentally accessed, such as by buffer overflow, the bot would self terminate as a protection mechanism. This was highly effective and led to a period of relatively low ban rates (2010-2012). However, significant ban waves in early 2013 made the community suspicious that Blizzard had a new method to detect bots. Eventually, further research led to the conclusion that Blizzard had in fact developed an algorithm that analysed character pathfinding behaviour to distinguish between bots and humans. This is done by recording a character's position over a period of time and by grouping closely clustered points into a single waypoint. The number of times the character moved between two waypoints was recorded and then a map of waypoints and path segments can be constructed.

The longest common prefix (LCP) for each character is recorded by calculating the longest repeated subpath a character has used. Because bots travel using a navigation mesh to move in straight lines between predefined waypoints, over a long running time they will inevitably start to repeat previously used path segments. Humans on the other hand do not have a defined navigation mesh and are unlikely to repeat exact movement patterns. A study done by researchers at the Technical University Vienna in 2009 showed that bots' average path segment repetitions and LCP tended to scale linearly with run time while that of humans stayed at a constant low rate [42] [48]. After around 10 to 45 minutes, there was a clear difference in LCP values between humans and bots.

2.5.6 Post August 2017

Finally, in August 2017, Blizzard released patch 7.3 for World of Warcraft, and with it the biggest security upgrade in the game's history. First of all, the game executable is now obfuscated, including the Warden anti-cheat system. This makes static analysis of the game much harder and in turn, harder to find pointer and memory locations. Difficulty of reverse engineering Warden itself means that successfully developed bots can no longer discover all of Warden's scan regions and properly implement a self termination defense mechanism. Blizzard uses the same type of obfuscation and anti-debugging methods across all of their games including World of Warcraft, Overwatch, Starcraft II. There has been a working Overwatch decryption tool produced in the past with positive reviews that has since been removed [6]. This lead me to conclude that it is possible to deobfuscate the World of Warcraft executable as well, although the methods to do so are not yet clear.

In addition, new anti-debugging measures cause the executable to crash when a user attempts to use a debugger. While some of Blizzard's anti-debugging countermeasures have been identified in the past, such as deliberately triggering hardware breakpoints causing debuggers to hang, not every technique implemented in the August 2017 patch is well understood. However, observations of the program consistently crashing when paused or resumed implies that a system clock check may be present. As of the current version, using a Vectored Exception Handling (VEH) debugger works, but the World of Warcraft executable detects the specific anti-debugging code injected by VEH debuggers and related programs. Detection of such abnormal behaviour results in the user's account being flagged for suspicious activity. This is identical to the technology first used in Overwatch to prevent debugging [7]. Because the technology is identical, using debug tools and scripts designed for Overwatch has been successful for dumping World of Warcraft [4] memory, which could yield useful information about Warden itself and lead to a functional debugger. Furthermore, because previously identified countermeasures have targeted user-level debuggers, it is possible that a kernel-level debugger could be used without detection or triggering program crash.

Finally, the update also implements trap memory segments. These are memory sections that look completely normal, but are monitored by Warden for external access. Attempting to perform a memory scan that accesses a trapped memory segment will either crash the client, flag the users account as suspicious, or both. It is difficult to determine which memory segments are trapped as the client is not notified of their account being flagged.

Due to Warden itself being obfuscated, it is currently not possible to determine which memory segments are 'trapped'. As a result, developers only discover they have accessed trapped segments when the account is suspended. Coupled with Blizzard's policy of banning users in periodic waves instead of immediately, bot developers cannot easily figure out which action caused an account loss.

These three most recent techniques have been considered very difficult to defeat and as

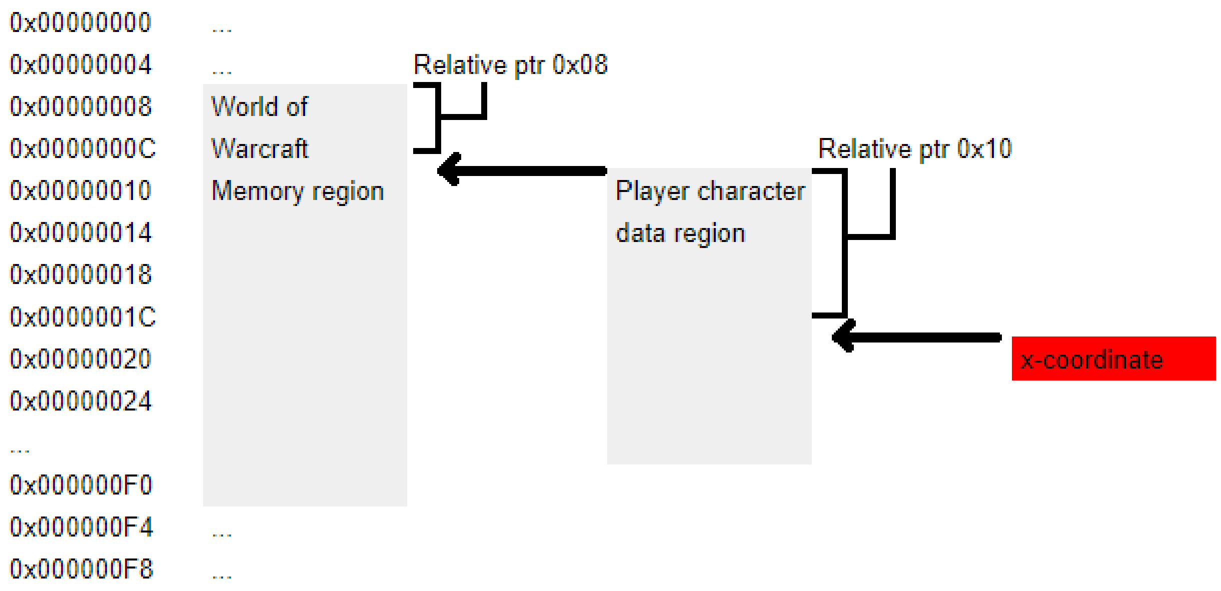


Figure 2.6: A simplified example of the WoW memory space.

a result both the major bot developers have shut down in response [8] [1].

2.5.7 Obtaining information

A bot or other game hack requires information from the game environment, such as player location or health values, in order to function. An intrusive bot keeps track of its current location in the game world by accessing the game's memory data. Because World of Warcraft stores game information in a fixed and predictable format, a bot can use a series of pointer operations (offsets) to access data. For example, conside Figure 2.6 which shows a simplified version of World of Warcraft's memory region and how the player's x-coordinate is stored in memory. In order to access the player character's x-coordinate, the bot needs to:

- Find the start of WoW's memory space within the computer main memory (also known as WoW base address) which is located at 0x00000008
- Read player character data by accessing memory at ”WoW base address → character data offset”. As this location is 0x08 bytes offset from the WoW base address, we can use a pointer reference to access the character data from the base address.
- Read player character x-coordinate by accessing memory at ”WoW base address → character data offset → x-coordinate offset”. As the x-coordinate address is 0x10 bytes offset from the character data region, we therefore get here by using a second pointer reference.

The use of multiple offsets to navigate to required memory addresses is made possible due to the ”object manager” data structure which as its name suggests, manages all other objects in memory such as player characters, items and enemies. This structure contains a mapping of references to all objects active in the game and can be used by external programs to determine offsets of dynamically generated objects. However, some static offsets, such as the base address of the object manager, are found manually through the use of debugging and memory reading tools. These offsets must be re-calibrated for each different version of the game as recompiling the executable will change memory locations.

2.5.8 Sending commands

After a bot successfully accesses the game's memory space and obtains the necessary game information, it must then issue commands to the game in order to control the character. Movement and interaction are performed by modifying values or calling functions from memory.

Human players move their characters within the game world through either the use of keyboard keys to issue relative movement actions (such as "move left"), or by right clicking on locations in the game world to issue a command for the character to move there ("click to move"). Relative movements require taking into account factors such as character facing direction, camera angle, relative distance to target, etc, while click to move commands only rely on the target's absolute coordinate. Therefore it is much easier for a bot to use the click to move system. When a player uses the click to move command, the following set of events occurs:

- Player clicks on a point in the world
- Click location is transformed into a set of XYZ coordinates by the game
- XYZ coordinates are written to a 'click to move target' variable in memory
- Game begins moving player character from current location to target in a straight line
- When player character reaches the target, game clears 'click to move target' variable

The bot, through direct memory access, can write the target coordinates straight into the games 'click to move target' memory address and begin moving the character.

When the bot detects that the player character is close to a resource node, it will attempt to gather from the node. In a normal human player's operation, the program flow is as follows:

- Player clicks on node to begin gathering
- Game recognises the click and calls a `BeginGathering()` function
- Resources from the node are deposited into the player's bag

Similar to writing data to memory, external programs are also able to call functions from memory if they know the base address of the function call when the program is loaded into memory. Thus, the bot is able to simply call the `BeginGathering()` function when it is near a node and gather resources without having to perform clicking. Although functions in WoW were loaded into the program memory at runtime, their addresses could be found by disassembling the game executable and statically analysing program flow by deducing their relation to static variables. For example, if the memory location of the player location variable was already obtained, one could obtain functions related to player movement by finding any functions that modified this memory location.

Chapter 3

Design and Implementation

3.1 Introduction

In this chapter we explore the design and implementation of the memory tool at the centre of this thesis. The memory tool's purpose is to simplify the process of reverse engineering the organisation of data in target programs and we will examine its use in some case studies later in Chapter 4.

First we will consider why the reverse engineering process needs simplification. Reverse engineering of unknown programs is slow because we often must manually discover required memory locations (also known as 'offsets') in an iterative fashion. For example if we wanted to know how to automatically move a player character in World of Warcraft, the memory address holding the "player location value" is of interest. Experimental testing has shown that many games, including World of Warcraft, uses a fixed memory allocation system [33] as previously shown in Figure 2.6.

In order to find this address, we might complete the following steps:

- 1. Move player to known location in game world, e.g. position with coordinates X = 1200, Y = 730, Z = 65
- 2. Iterate over the game's memory space and create a list of all memory locations containing the numerical value "1200"
- 3. Move player to another known location in the game world, e.g. position with coordinates X = 1300, Y = 730, Z = 65
- 4. Intersect the lists created in steps 2 and 3, thus creating a list of memory locations whose values have changed from 1200 to 1300.

Steps 3 and 4 must be repeated many times in order to narrow down the address to a single location for each value we are interested in. In the example of automating player character movement, this means we have to repeat the entire process for the X, Y and Z coordinates. As one can imagine, this takes a significant amount of time and effort to build a list of memory locations in larger reverse engineering projects. Therefore our goal is to create a tool that assists in automating parts of the reverse engineering process. We used a simple Windows game, Minesweeper, to test this tool.

3.1.1 Design

The tool consists of memory dumper and dump analysis modules that work together to analyse a target program's operation by automating the analysis described previously. The

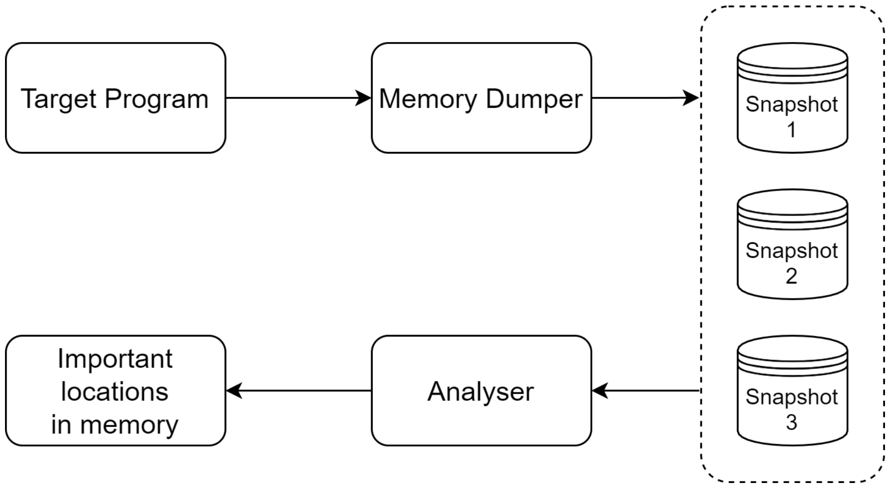


Figure 3.1: System architecture diagram of the tool showing the important components. The memory dumper saves the target's memory to disk. The analyser takes the snapshots created by the dumper and compares them to determine important locations in memory

user interacts with the target program in a consistent manner and the memory dumper saves the target's memory data to disk as a series of snapshots. The analyser then reads memory files and compares them to locate differences. This process is shown in Fig 3.1.

After the memory dumper has created a number of snapshots with consistent changes between them, we can then use the analyser. The analyser detects differences between snapshots and given enough snapshots, it will be able to narrow down the locations in memory which have consistently changed and mark them as important.

Key aspects of the design were a focus on tool efficiency, effectiveness and usability. We used an iterative approach to optimise the tool in all three areas over the course of its development. Due to the number and size of snapshots, iterating over each pair of snapshots to find differences is computationally intensive. As a result, we had to optimise our tool to complete this process efficiently to be able to perform a meaningful amount of testing.

In addition to efficiency, we had to ensure the tool was effective by taking measures to eliminate false positives (noise) that cluttered the results. To achieve this, we implemented an effective algorithm to detect and isolate memory regions that consistently changed as a result of consistent user input. This filtered out the noise and thus made the analysis more effective.

Finally, in order to be easily usable, we designed the tool's output to be easy for the user to understand. Initial versions used a visual representation of the target program's memory which proved to be hard for the user to read. By the final version, the output had evolved to become a simple text output showing the address and data of important memory regions.

3.2 Reverse engineering Minesweeper

Minesweeper is a popular puzzle game shipped with the Windows Operating System since 1992 [22] where the player's goal is to find all hidden mines in a grid of squares. A player must use spatial logic, and the surrounding numbers to determine where the mines are in the grid. The player can then right click the squares containing suspected mines to mark them with flags, or left click on squares they believe to be empty to reveal them. Successfully revealing all squares devoid of mines and deducing the location of all mines in the grid results in winning the level, while an incorrect use of logic results in a player exploding one of the mines, immediately losing the level and having to begin the entire level again. There are many possible goals for reverse engineering this game, such as:

- Finding squares with hidden mines easily
- Marking and unmarking mines automatically without user input

The primary goal of reverse engineering this game is to assist the user in determining the relationship between an action and the memory space it affects. One approach involves taking a snapshot of the target program, performing an action that changes the program state and then taking another snapshot. The program will then take the difference of the two snapshots to find memory locations that changed after the action was performed. We have four assumptions heading into the experiment:

- A user action causing a change in program state must also cause a change in memory. These changes in memory are useful to the engineer.
- Not every change in memory must be caused by a user action. These changes are not useful and are considered noise.
- Statically allocated memory locations for a program will remain at a consistent and fixed location between runs. This assumption is made because programs allocate memory at runtime and generally make no attempts to randomise memory locations. This is because randomising memory locations involves also updating pointer references to each one and is too costly to be beneficial. While certain exceptions exist, such as Denuvo anti-tamper software which repeatedly encrypt and decrypt programs at runtime, they are rare and we will not be encountering them in this experiment.
- Dynamically allocated memory may remain at a consistent and fixed location under some special circumstances particularly in cases when memory is allocated in exactly the same order on every run. While malloc and dealloc operations may be deterministic to a degree [57] [53], this is likely to not be the case in concurrent applications [46] or those involving unpredictable user input. In such cases, the order in which memory, and as a result the location of of certain memory data, is likely to change.

An example of such cases would be changes in the game state of minesweeper as shown in Figures 3.2 and 3.3. In Figure 3.2, we can see an example of the first case where the user has right clicked on the circled tile and caused it to be marked with a flag. In the game's memory space, a value corresponding to the state of that tile has changed. Finding the specific locations of this value in memory is important for finalising the relationship between the action ("right click on the tile") and memory location affected. On the other hand, consider Figure 3.3 where there has been no user action, yet the timer value still changes. This is noise because regardless of what user action was performed, the memory location storing the timer value will always change. It is not very useful to identify this location as a result of user actions despite it changing every time we perform an action.

Because program state and memory data can change without external output, we must take many snapshots in order to filter out such noise and leave us with only the relevant memory data. Therefore the tool must take snapshots at regular intervals while the user performs actions in a consistent way.

Furthermore, because consistent, unrelated events such as timers will cause noise, we also must use certain noise reduction techniques to filter this form of noise out over the course of several experiments. Our main strategy was comparing sets of experiment runs based on the same user action until there is a clear set of consistent memory changes.

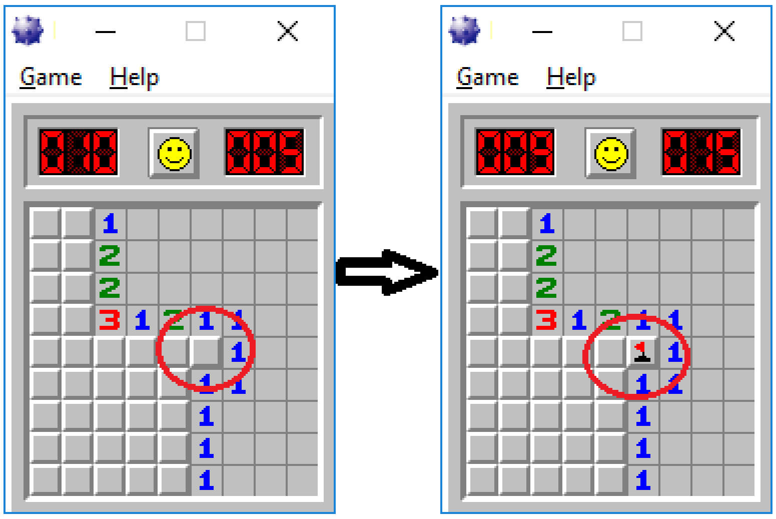


Figure 3.2: An example where an outside action causing a change in program state (the circled tile has been flagged) causes a change in program memory.

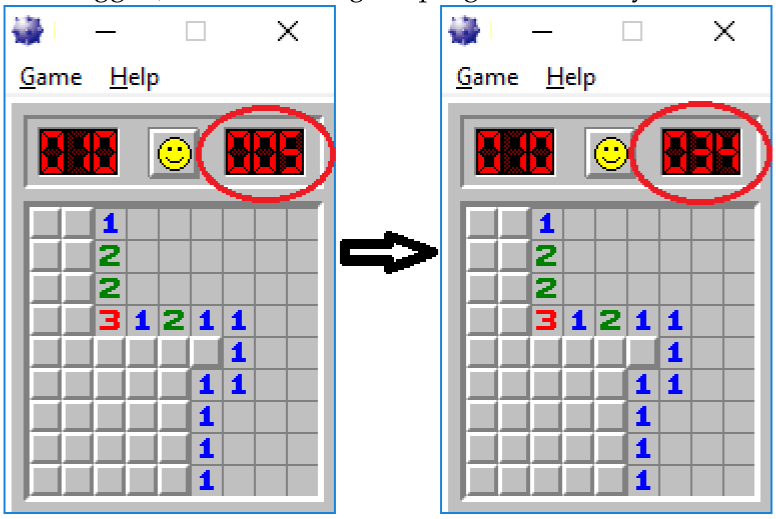


Figure 3.3: An example where no outside actions have occurred, yet there is still a change in program state (the timer clock value has increased) and program memory.

3.3 Iteration 1

The first prototype was designed to be a helper tool that could find memory locations of interest and assist in understanding the purpose of these locations. It was written in Java as a starting point and made extensive use of the JNA (Java Native Access) library. Java runs inside a Java Virtual Machine (JVM) and normally cannot access low level information such as memory data of other processes as they lie outside the scope of its own JVM instance. JNA essentially functions as a wrapper that allows the user to write applications in pure Java and make use of native Windows functions without having to directly call Windows API functions. The JNA library handles calling the equivalent Windows API functions when necessary which means the user can avoid using C, via the Java Native Interface, within their Java application. Because our experiment relied on reading and writing to the memory space of other programs, JNA was essential for providing us access to Windows API calls that can fetch this information. For example, two Java methods provided by JNA which

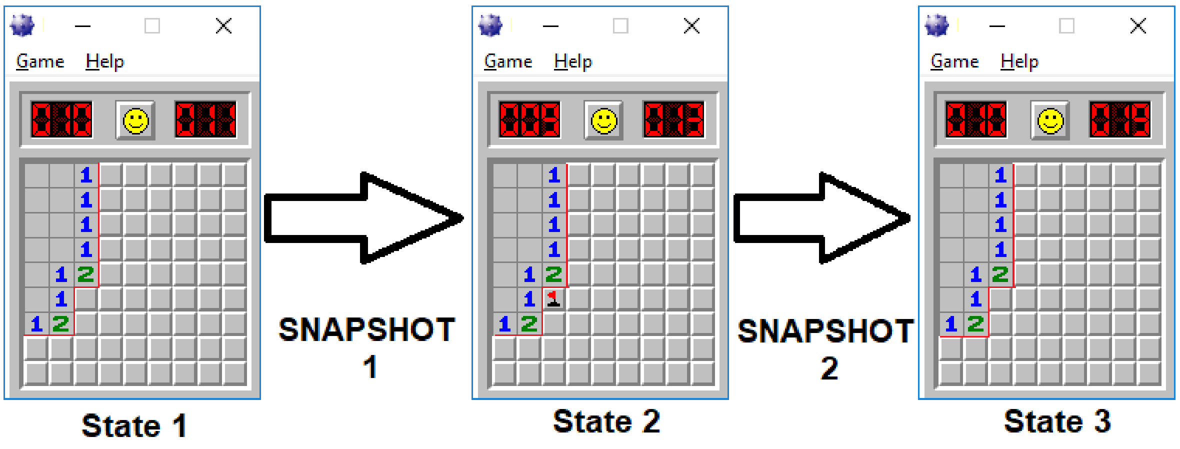


Figure 3.4: A simple example of a user interacting with the minesweeper game while the tool takes snapshots. The red line indicating the boundary of the 'exposed' region has been added for comparison to information below.

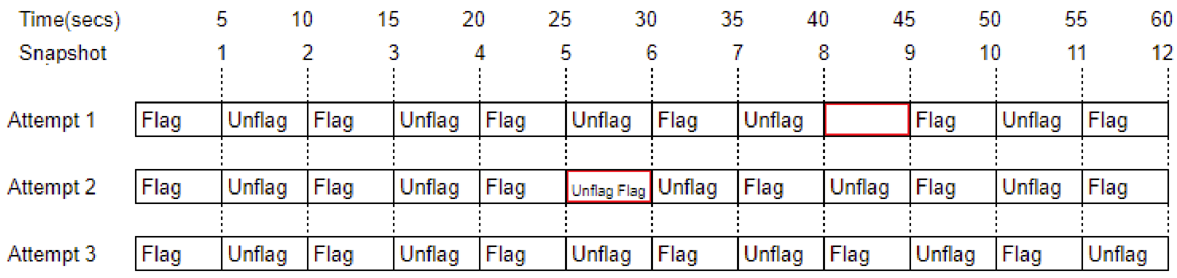
were used extensively in this prototype were:

- `getProcessID`, equivalent to `GetProcessesByName` in the Windows API, which obtains a PID(Process ID) of a Windows process based on a window name.
- `openProcess`, equivalent to `OpenProcess` in the Windows API, which opens an active process by PID as an object and allows reading or writing to its memory space.

The Java tool simply took snapshots of the target program memory at regular intervals of 5 seconds over a minute period making a total of 12 snapshots per run while the user interacts with the target program in a consistent, repeated manner. It was believed that such consistent interactions would generate consistent results which then could be analysed later to identify changes. For example, if the user repeatedly flagged and unflagged a single tile in the minesweeper game, then in theory only the memory location corresponding to that square would change. After snapshots were taken, the tool would then perform an intersection operation over all snapshots and output an image file representing changes detected where each pixel represented a single byte of memory. Shading of pixels represented the number of intersections where the value of that pixel changed, with darker pixels signifying more changes. This was useful as a preliminary experiment because we were unsure about the magnitude of memory changes resulting from a single action and thus wanted a visual representation that provided the user with an overview of the number and location of changes at a glance. The user could then use this information in conjunction with other tools(such as Ollydbg) to analyse the actual data changes.

A simple example of using the tool on minesweeper is shown in Figure 3.4. The user repeatedly flags and unflags the tile in sync with the tool taking snapshots at set intervals. Because the tool was originally calibrated to perform one snapshot every 5 seconds, this meant that the user had to ensure they performed the correct action between snapshots or risk the experiment being contaminated by user errors. For example, consider Figure 3.5 which demonstrates three different possible scenarios encountered by a user attempting to find information regarding a certain tile, "X", within Minesweeper by repeatedly flagging and unflagging this tile while the Java tool takes snapshots:

- In attempt 1, the user was too slow to perform the flag action that was expected between 40-45 seconds into starting the program. As a result, between snapshot 8 and 9,



Time(secs)	5	10	15	20	25	30	35	40	45	50	55	60
Snapshot	1	2	3	4	5	6	7	8	9	10	11	12
Attempt 1	Flag	Unflag	Flag	Unflag	Flag	Unflag	Flag	Unflag		Flag	Unflag	Flag
Attempt 2	Flag	Unflag	Flag	Unflag	Flag	Unflag Flag	Unflag	Flag	Unflag	Flag	Unflag	Flag
Attempt 3	Flag	Unflag	Flag	Unflag	Flag	Unflag	Flag	Unflag	Flag	Unflag	Flag	Unflag

Figure 3.5: Examples of correct (3) and incorrect (1 and 2) operation of the basic Java tool.

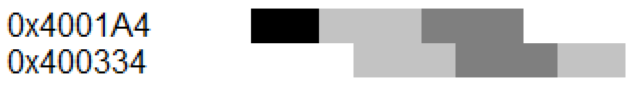


Figure 3.6: An expanded and annotated section of the final output displayed by the program where darker pixels represent locations in memory that have changed more often and lighter pixels represent locations that have changed less often.

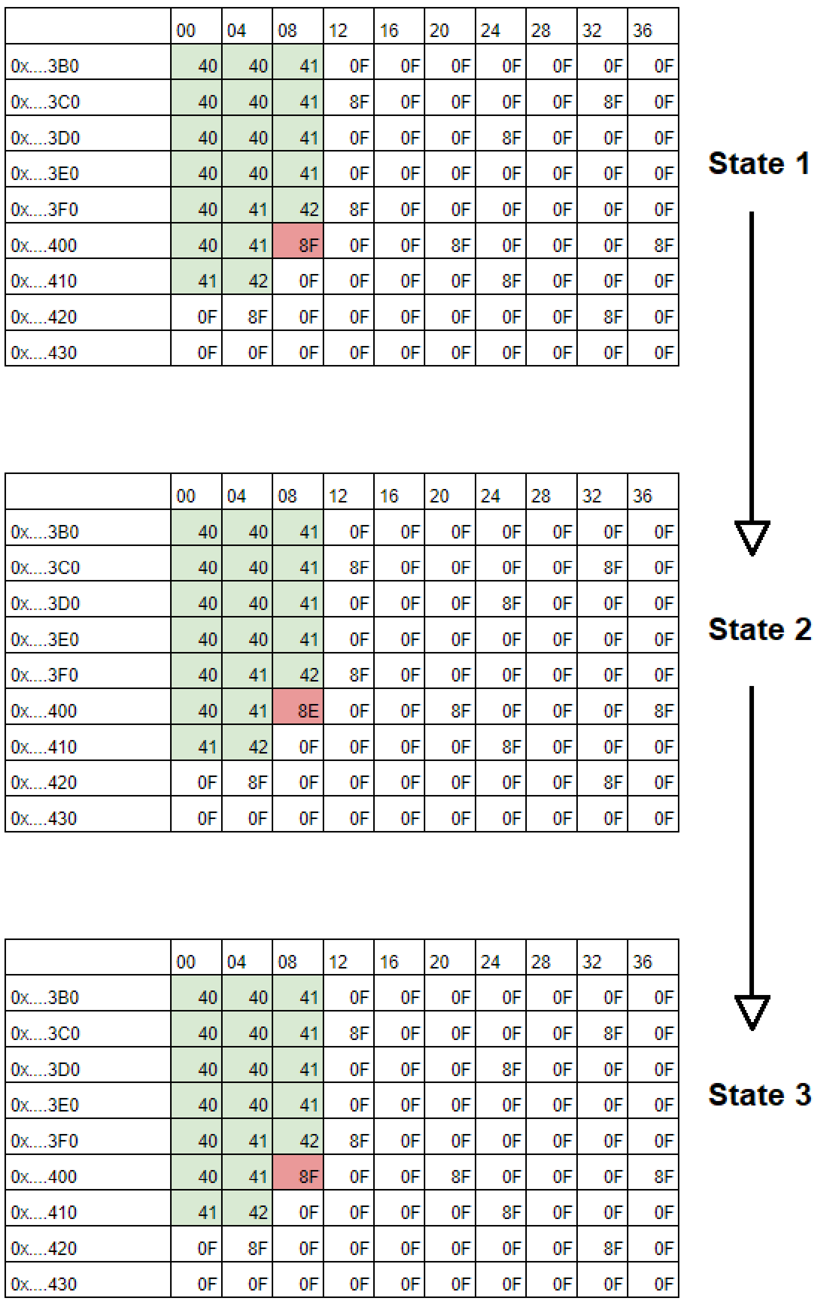
there is no change to tile X's value when the program expects one. Therefore the 'true' memory location of tile X would be excluded from the experiment results as it did not change when the program expected such a change.

- In attempt 2, the user was too quick to perform the flag action before snapshot 6 was taken. As a result snapshot 6 was taken with tile X in a 'flag' state, same as snapshot 5. This results in the same issues with experiment results as attempt 1.

- Finally in attempt 3, the program and user behaved as expected, with exactly one action performed between each snapshot.

While the tool required precise timing from the user to avoid problems, it achieved the desired function of helping with reverse engineering when used correctly. An example of the output is displayed in Figure 3.6 where darkest pixels represent memory locations exhibiting the most changes. By identifying the regions with significant numbers of changes, I was able to then inspect the values at these locations with tools such as Ollydbg while the program ran to understand how and why they changed. Without the tool narrowing the search region down, it would have required significantly more work to manually browser the entire memory space instead of just the small areas that changed.

Figure 3.7 shows a small section of the game's memory changes during this process. The circled value alternating between 8F and 8E represents the 'flagged' tile and is the key information we wish to discover. The memory addresses corresponding to the area of change were found and examined in Ollydbg to view the specific contents at those locations. It was found that the memory addresses belonging to some of the darkest pixels correlated with the region in memory where the game board is stored. In other regions of the image output there existed noise as there are many other regions of dark pixels which are memory locations which changed, but not directly as a result of user action. For example, the variable containing time since game start continues to change as we perform the experiment but is



State 1

	00	04	08	12	16	20	24	28	32	36
0x....3B0	40	40	41	0F	0F	0F	0F	0F	0F	0F
0x....3C0	40	40	41	8F	0F	0F	0F	0F	8F	0F
0x....3D0	40	40	41	0F	0F	0F	8F	0F	0F	0F
0x....3E0	40	40	41	0F	0F	0F	0F	0F	0F	0F
0x....3F0	40	41	42	8F	0F	0F	0F	0F	0F	0F
0x....400	40	41	8F	0F	0F	8F	0F	0F	0F	8F
0x....410	41	42	0F	0F	0F	0F	8F	0F	0F	0F
0x....420	0F	8F	0F	0F	0F	0F	0F	0F	8F	0F
0x....430	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F

State 2

	00	04	08	12	16	20	24	28	32	36
0x....3B0	40	40	41	0F	0F	0F	0F	0F	0F	0F
0x....3C0	40	40	41	8F	0F	0F	0F	0F	8F	0F
0x....3D0	40	40	41	0F	0F	0F	8F	0F	0F	0F
0x....3E0	40	40	41	0F	0F	0F	0F	0F	0F	0F
0x....3F0	40	41	42	8F	0F	0F	0F	0F	0F	0F
0x....400	40	41	8E	0F	0F	8F	0F	0F	0F	8F
0x....410	41	42	0F	0F	0F	0F	8F	0F	0F	0F
0x....420	0F	8F	0F	0F	0F	0F	0F	0F	8F	0F
0x....430	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F

State 3

	00	04	08	12	16	20	24	28	32	36
0x....3B0	40	40	41	0F	0F	0F	0F	0F	0F	0F
0x....3C0	40	40	41	8F	0F	0F	0F	0F	8F	0F
0x....3D0	40	40	41	0F	0F	0F	8F	0F	0F	0F
0x....3E0	40	40	41	0F	0F	0F	0F	0F	0F	0F
0x....3F0	40	41	42	8F	0F	0F	0F	0F	0F	0F
0x....400	40	41	8F	0F	0F	8F	0F	0F	0F	8F
0x....410	41	42	0F	0F	0F	0F	8F	0F	0F	0F
0x....420	0F	8F	0F	0F	0F	0F	0F	0F	8F	0F
0x....430	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F

Figure 3.7: A visualisation of the data structure (array) containing the game board in memory. Note that the green region represents the 'exposed' region and that states 1 and 3 are identical. The 'flagged' tile has changed from value 8F to 8E and back to 8F.

unrelated to user input. As a result, the memory region containing the clock variable as well as any graphical assets used to render the clock will continue to change and be detected as a dark region in the image output even though it is not very useful for the experimental purpose of locating the memory region modified by a user clicking on the game board.

The tool mostly achieved its purpose and helped to clarify the data structure for certain programs such as minesweeper and confirmed the presence of a fixed region in memory containing the game board. While it did suffer from noise by locating regions that were not interesting, the key regions were properly displayed as dark areas in the program output. In addition, a consistently changing memory location corresponding to the tile being clicked on was located by analysing the image output. Locations of most change (darkest regions) were identified and the memory locations were opened in Ollydbg for manual analysis. As a result, the value of a single memory location changing between values of '8F' (no flag) and '8E' (flag) was attributed to the user clicking on the tile and changing its state.

However the prototype could not be used as a standalone tool and was often confusing due to significant noise in the output image. In addition due to a lack of optimisation, long run times of up to several hours ensured that only a limited number of tests could be run on a given day. Due to these issues it was clear a better solution was required.

3.4 Iteration 2

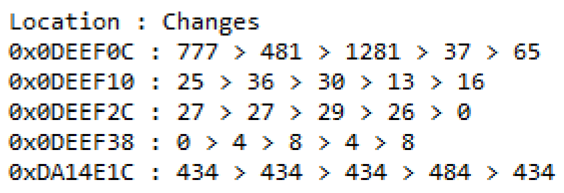
The first iteration was a good starting point that correctly identified locations where data changed and supported further analysis of programs. However, two major problems with the execution and output were identified and improved upon in our second iteration:

- Usability of the output data
- Program execution speed

3.4.1 Usability

The program output shown in Figure 3.6 was difficult to decipher as it lacked detailed information such as how values in memory were changing. As a result, the program by itself was not able to easily connect a user action with a memory location and as a result overall program usability suffered. The concept of a visual representation was useful to identify rough areas of change in memory but was not specific about the exact location or values of change. As a result, while it was successful for visualising the 'big picture' in a preliminary experiment, it was not particularly useful for obtaining the exact data needed for further tests. Furthermore, because large amounts of program memory did not change, the user must look over large amounts of white space to identify useful sections. This would prove to be further problematic with larger programs as image sizes quickly became unmanageable sizes when even a relatively small program using 10MB of memory generated a large image file of approximately 2000x5400 pixels.

In order to avoid these issues, the previous image based output was abandoned and instead the location and values of memory changes were displayed directly. A sample output of this is shown in Figure 3.8. By changing the output to display numerical values at each memory location, the program was also able to now be used independently of other programs to determine patterns of change in memory. While this approach appeared to run into the original problem of requiring the user to pore over large lists of meaningless data including noise, this was in fact not the case as relatively few locations changed. Finally, by having access to exact memory locations and values, it provided a good foundation for the final version of the program.



```
Location : Changes
0x0DEEF0C : 777 > 481 > 1281 > 37 > 65
0x0DEEF10 : 25 > 36 > 30 > 13 > 16
0x0DEEF2C : 27 > 27 > 29 > 26 > 0
0x0DEEF38 : 0 > 4 > 8 > 4 > 8
0xDA14E1C : 434 > 434 > 434 > 484 > 434
```

Figure 3.8: A sample of the tool output showing how changes in value of memory locations are displayed.

3.4.2 Execution speed

The first version focused on getting a minimal viable product (MVP) running without optimisations. As a result, it ran slowly, often taking an hour or more to complete a full experiment analysis based on just one minute of a target program's runtime. The slow execution speed was largely caused by inefficient use of loops and data structures in the first prototype. The slow runtimes, compounded by mistakes in user operation, meant that often only a few meaningful experiments could be conducted in a day. Because large numbers of experiments must be performed to get accurate data, it was clear program runtime had to be increased significantly. Two potential solutions were considered in response to this issue that both had the same goal of reducing time taken per experiment run:

- Increasing effective processing power
- Increasing program efficiency

Increasing effective processing power, specifically how much memory information could be analysed in a given time with constant hardware, was explored first as it was believed that this could provide the biggest improvement in execution time with relatively few changes to the program code. Our primary solution was a chunk based approach to the data analysis done by splitting the entire data dump into a number of smaller, more manageable chunks. Each chunk in the first dump was then compared to its corresponding chunk in the second one. By splitting the workload into smaller chunks we were able to multi thread the process for additional improvements in speed.

At the same time, the following steps were taken to increase the programs execution time on a single computer by optimising the code. A key issue that slowed down program execution was identified as the use of many Java objects, such as 'Byte', rather than their primitive counterparts, 'byte'. Because JNA used object based data structures such as `ByteArray`, this apparently minor difference was overlooked in the first iteration leading to most of the code using objects instead of primitives. This turned out to be a significant slowdown in performance as Java has more overhead when using objects compared to using primitive data structures. Compounding this was the fact that it simply takes longer to load and process larger objects in memory. For example, when comparing and processing a large array of Bytes, each byte value has to be retrieved from the Byte object through the use of getter methods before it can be compared, an issue that is not present when simply dealing with arrays of primitive bytes.

Most operations and data structures using Java objects such as Byte's were therefore rewritten to instead use primitives such as bytes in order to speed up program runtime. This was a big improvement in execution speed, with run times dropping to as low as 5 to 10 minutes for the same experiments that once took over an hour.

Finally, for additional speed improvements, the entire program was ported to C#. Because at this point the program largely used primitive data types, the code itself was similar

	User action	No user action
Consistent change	Useful	Not useful
Random change	Not useful	Not useful
No change	Not useful	Indeterminate

Table 3.1: A summary of possible user action and memory change combinations and whether the result is helpful in determining the user action to be the cause of the memory change.

enough to C# that it could be ported without much major changes. By using C#, we had access directly to the low level Windows functions previously accessible only through the JNA wrapper and was no longer slowed by the internal operations of the JVM. This further sped up execution to around 3 to 5 minutes per experiment and also had the additional benefit that C based Windows functions were much better documented than their JNA counterparts, leading to easier coding and debugging.

3.5 Iteration 3

While the upgrades added in iteration 2 significantly improved both the program runtime and readability of the final result, it still suffered from noise due to memory values in target programs changing because of reasons other than user input. Noise included memory locations such as the ones keeping track of 'time elapsed' as these would change every time a snapshot was made even though user input is not what causes a change in time elapsed. Because there was noise in the output, it was difficult to determine exactly what changes were caused by the user. In addition, the analysis process was still reliant on user timing and manual interpretation of the results. As a result additional improvements were needed to filter out this noise and make the analysis process more streamlined.

3.5.1 Noise reduction

Up to and including iteration 2, noise reduction had consisted of an iterative process of identifying regions which change when a user action is performed. Regions which did not change after a user action were filtered out as irrelevant. While this approach meant that all useful regions were identified in the final results are useful, it also included many irrelevant regions in the results as well.

Consider Table 3.1 which describes possible categories of user actions and changes in memory caused by them between snapshots. The six possibilities are :

1. User performs an action and a memory region changes consistently as a result of the action. This is the key useful case as it immediately tells us that region in memory is linked to the action we performed. An example of this is user flagging a tile in minesweeper, and we detect a change in the tile's memory state.

2. No user action and a memory region changes consistently. This is observed to occur in memory regions that have no relation to a user action. For example, the memory location containing a timer value changes as a user idles.

3. User performs an action and a memory region changes inconsistently. We cannot directly establish a relationship between the user action and memory region

4. No user action and a memory region changes randomly.

Example	Location	Experiment 1 Snapshot 1	Experiment 1 Snapshot 2	Experiment 2 Snapshot 1	Experiment 2 Snapshot 2
1	0x00034000	8E	8F	8E	8F
2	0x00034040	0F	1F	1F	2F
3	0x00034080	0F	1F	2F	2F

Table 3.2: Comparing two experiment runs to detect consistent change.

5. User performs an action and a memory region is not changed. Not useful as clearly the memory region is not related to the user action being performed

6. No user action and a memory region remains unchanged. The memory region in question is of indeterminate usefulness and more information is needed to assess its relation to user actions.

For the purposes of establishing direct relationships between user actions and affected memory regions, cases which fall into category 1 are of the most interest to us. However in preliminary experiments, categories 2, 3 and 4 are also included in results due to the the program recording any memory region that experienced a change, without the ability to distinguish between consistent and random changes. Furthermore, because the experiments lacked a phase where no user input was expected, it was not possible to distinguish between changes caused by user actions(categories 1,3) and changes that would have occurred regardless of user actions(categories 2,4). The third iteration would need to solve these shortcomings by using additional noise reduction strategies.

To solve these problems, we improved the strategy of intersecting sets of snapshots by looking at not just what memory locations changed, but also how values changed between snapshots. In addition, we further identified regions of memory that consistently changed after a user action is performed. This resulted in obtaining a definitive set of memory locations that changed predictably in response to user actions.

This approach involved creating pairs of game state dumps that only differed by one game action, such as flagging and unflagging of a single minesweeper tile. These pairs of dumps were then compared and the difference stored on disk. After a number of differences were recorded, they were compared to differences of other pairs and matching changes between the differences were recorded. By using this method of intersecting snapshots, only addresses that changed consistently were identified, i.e. addresses that changed each time between the two game states, but the change was the same between each test.

For example consider Table 3.2 where three memory locations which are found to have changed during two experiment runs of the program where the user performed the same action of marking the same tile in the Minesweeper game board.

While each memory location's value did change during the experiments, only example 1 displayed a consistent change in both experiment runs given consistent input actions. That location, 0x01005362, consistently changed between 8E and 8F. Therefore, we can hypothesize that this memory location is directly affected by the user action.

3.5.2 Analysis improvements

The most important change was that the snapshot system was improved to be more user friendly and easier to analyse. The previous versions required precise user timing which sometimes caused problems. Furthermore, the user had to manually record which actions were used in each experiment run. We aimed to fix both of these issues in the final version.

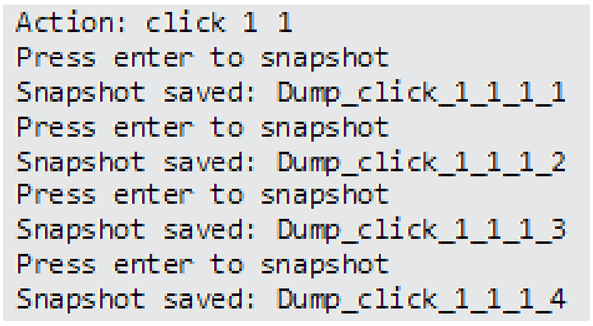

```
Action: click 1 1
Press enter to snapshot
Snapshot saved: Dump_click_1_1_1_1
Press enter to snapshot
Snapshot saved: Dump_click_1_1_1_2
Press enter to snapshot
Snapshot saved: Dump_click_1_1_1_3
Press enter to snapshot
Snapshot saved: Dump_click_1_1_1_4
```

Figure 3.9: Example of console output of new tool where user dictates when snapshots are taken. User must input the type of action performed (in this case ”click 1 1”) before commencing the snapshot process. A snapshot is only taken when the user sends input to the program by pressing enter.

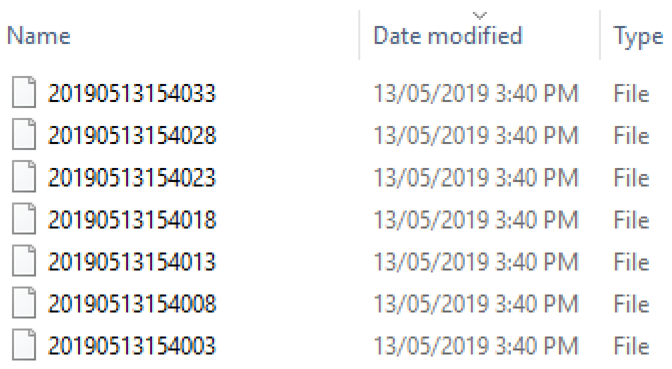
First of all the user was given more control over the snapshot taking and saving process. Prior to an experiment run, the user inputs the name and type of the action to be tested for. The user then interacts with a target program as before, but the tool only took snapshots when the user desired, i.e. by pressing a key on the tool’s console. The snapshot would then be automatically saved to disk with the specified action encoded in its name. An example of a typical program run is shown in Figure 3.9.

While this was a minor change, it gave the user better control over the experiment process and allows better snapshot management. The old system saved memory dumps to file based on a timestamp format, as we can see in Figure 3.10. This worked fine in the initial experiments as memory dumps were not intended to be reused. However as we now wished to preserve experiment data for future use, a new storage method was required.

This was changed by encoding the user action into the name of saved snapshots. Before starting the recording process, a user must enter the the name of action being examined, such as ”click 1 1” when experimenting with the user action of ”click on the tile at row 1 and column 1”. Once the test action was defined, the program can then be commanded to take snapshots when needed. An example of the program console is shown in Figure 3.9. The resulting snapshots are saved to disk with the experiment details saved in the filename. The new storage format, shown in Figure 3.11, is more useful as it records the user action (”click at location 1-1”) as well as experiment number (1) and snapshot numbers (1-4). Afterwards, the program would then parse both the file data as well as the file name to establish a connection between the action taken and changes in memory.

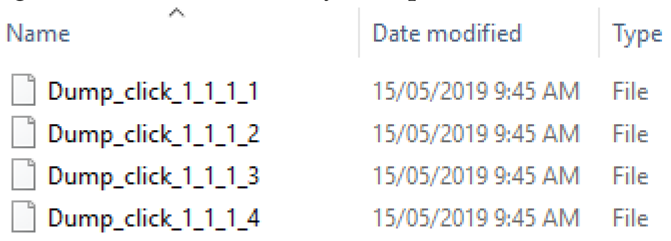
Finally, we were able to automate the analysis process further as well as preserve the data in its original form for future processing due to the improved naming format. Formerly in memory, the recorded snapshots were stored in a data structure consisting of a collection of byte arrays, with each array containing a single snapshot of the memory. In the new implementation, instead of working with simple arrays, we instead have a collection of MemoryDump data structures as follows:

```
public struct MemoryDump
{
        public String test_type;
        public int test_num;
        public int snapshot;
        byte [][] data;
}
```



Name	Date modified	Type
20190513154033	13/05/2019 3:40 PM	File
20190513154028	13/05/2019 3:40 PM	File
20190513154023	13/05/2019 3:40 PM	File
20190513154018	13/05/2019 3:40 PM	File
20190513154013	13/05/2019 3:40 PM	File
20190513154008	13/05/2019 3:40 PM	File
20190513154003	13/05/2019 3:40 PM	File

Figure 3.10: Saved memory dumps under old format



Name	Date modified	Type
Dump_click_1_1_1_1	15/05/2019 9:45 AM	File
Dump_click_1_1_1_2	15/05/2019 9:45 AM	File
Dump_click_1_1_1_3	15/05/2019 9:45 AM	File
Dump_click_1_1_1_4	15/05/2019 9:45 AM	File

Figure 3.11: Saved memory dumps under new format

By allowing the tool access to information regarding the details of the test, it was able to automatically connect specific memory locations to actions that affects them. For example, previously after an experiment the tool displayed `"Location 0x00034000 change 8E -> 8F"`. With contextual information from the user it was able to more clearly output information such as: `"Click 1 1 causes location 0x00034000 change 8E -> 8F"`.

Furthermore, these changes in the analysis process in addition to the improved data storage system allows us to store large amounts of snapshots for mass analysis at a later time. As each snapshot is encoded with the experiment type and numbers in its filename, we can easily serialise and deserialise snapshots stored on disk back into its in memory representation. This is important as it greatly increases the level of automation of the program and potentially allows future analysis methods to be used for data we record today.

Chapter 4

Case Studies

We will now discuss the process used to evaluate the tool developed. The purpose of the evaluation was to assess how useful the tool was in assisting a user to perform reverse engineering of a target program. It was hoped that the tool would expose information about the target programs' data structures in memory. To assess its usefulness, we tested the operation of our tool on three different programs and recorded the memory locations and data it reported. Finally we verified the accuracy of this information by examining the recorded memory locations in target programs and observing how the data changes during program operation.

4.1 Experimental Method

To test our tool's usability, we employed it on a small selection of real world Windows applications. Windows was chosen as our target platform as it is the most common desktop operating system and the platform our tool was compiled for. Furthermore, we decided to target applications with a graphical user interface as these programs exhibited a visible and immediate response to user actions.

Based on these criteria, we chose Minesweeper, Notepad++ and Solitaire as our test targets. These programs not only fit our criteria of Windows GUI applications, but also were straightforward to understand. For example, directly clicking on a square in Minesweeper would produce a discernible change in the game state which both the user and memory tool can detect. Furthermore, these programs have obvious differences in their functionality, thus providing a more representative spread of programs . For example, while Minesweeper has a grid based user interface and thus likely to have an array based memory layout, Notepad++ and Solitaire are not as obvious and thus required greater examination using the tool to discern their memory structure.

For each case, the memory tool was used to take snapshots of the target program while controlled user actions were performed. These controlled actions were carefully picked to exercise important features of each program and were common actions performed by everyday users. In addition, each action being tested caused a visible change in the program which can be associated with a change in the program's memory state. The actions were:

- Minesweeper : Flagging and unflagging of a tile
- Notepad++ : Text input, moving the input cursor and highlighting a region of text
- Solitaire : Moving a card in the playing field and flipping (revealing) a hidden card

After enough iterations had been performed, defined as when the memory tool no longer saw a change in the number of affected memory locations, the experiment was stopped. The affected memory locations and data would then be analysed and verified using a third party tool.

For all experiments, we used a desktop PC running Windows 10 Home Edition 64-bit. This machine had 32 gigabyte of RAM and a quad core i7 6700k CPU at 4.2GHz. For verification, we used OllyDbg v1.10.

4.2 Minesweeper

Minesweeper is a popular puzzle game shipped with the Windows Operating System since 1992 [22] where the player's goal is to find all hidden mines in a grid of tiles. Players can right click a tile to mark it with a flag indicating the suspected location of a mine or left click a tile to reveal it. Successfully revealing all non-mined tiles and marking all mines results in a victory while revealing a tile, through a left click action, containing a mine results in an immediate loss. The objective is to win the game by correctly identifying all mines in the shortest amount of time.

Each tile has a number of distinct states such as:

- An unrevealed tile which could be either a mine or empty space.
- A revealed empty tile which shows the number of mines near it. Note a revealed mine causes the game to end immediately.
- A tile with a flag on it placed by the player showing that they believe there is a mine in this tile.

The game itself has other functions such as a clock keeping track of the time spent on the current level as well as a high score system keeping track of the player's best playthroughs defined as victories in the shortest amount of time.

4.2.1 Goals of the analysis

Minesweeper was picked first to test the memory tool out on a realistic example of a video game. While it is simplistic in comparison to other games, Minesweeper contains many of the key features of computer games such as a dynamic game world and game states that change as a result of user input. By testing the automatic reverse engineering on Minesweeper, we simulate the thought process used by reverse engineers on more complex applications and are able to determine whether or not this process would work on a larger scale. The main goals of the analysis were to obtain the following information:

- Locate the data structure containing the game board in memory.
- Determine how the different states of tiles are stored in memory, e.g. unrevealed vs empty vs flagged
- Determine location of mines before they are revealed.

We were interested in seeing if it was possible to extract key information from the game as it would show that the reverse engineering process worked and allows it to be used on larger programs. For example, if we were able to locate mines before they are revealed in Minesweeper, another reverse engineer using this process may be able to reveal his opponent's units in a larger real time strategy game.

Furthermore, we were interested in seeing if our results were consistent over not just several experiment runs, but also program executions. This was useful to know as it could provide hints on how data structures are created at runtime.

4.2.2 Method

As Minesweeper is no longer included in Windows 10 we used Winmine [9], a 2001 version of Minesweeper bundled with Windows XP to perform tests on. To perform the experiment we ran the Minesweeper game and gave it controlled user actions while snapshots were taken with the memory tool.

After launching Minesweeper, a typical experiment run involves:

- Use memory tool to take snapshot of game memory with no tiles flagged.
- Flag a square (for example at x=1,y=1).
- Use memory tool to take snapshot of game memory with tile flagged.
- Remove the flag that we previously placed, returning the game to its initial state so we may repeat these steps.

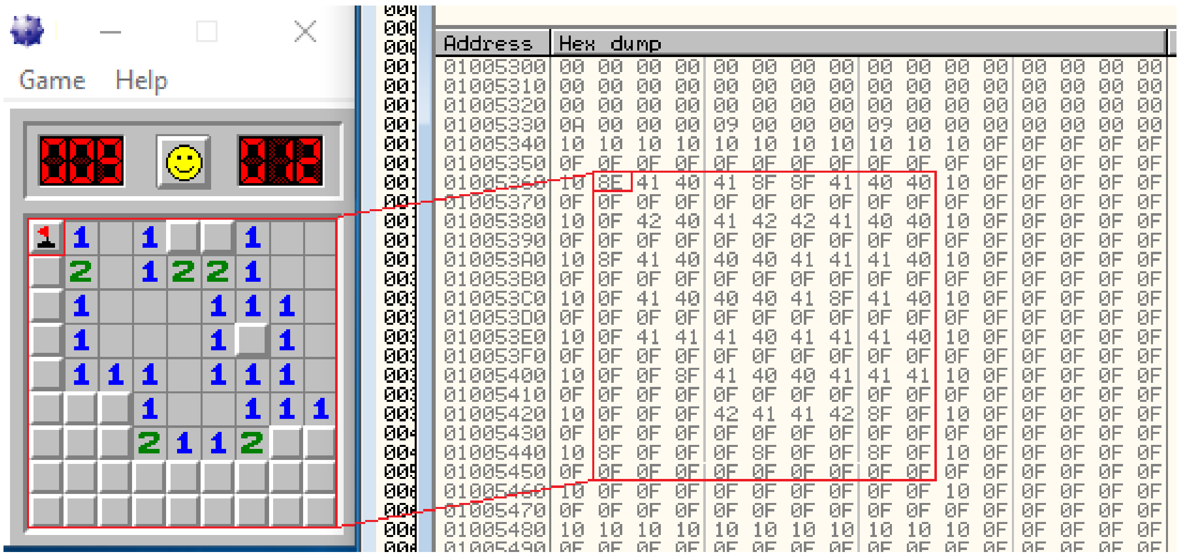
In order to get a good sample size, five locations on the game board area were tested:

- x=1, y=1
- x=1, y=2
- x=1, y=4
- x=2, y=1
- x=4, y=1

The flag and unflag process was repeated for each of these locations five times for a total of ten snapshots for the memory tool to analyse. This number of experiment runs was picked as previous preliminary experiments showed that three to four runs were sufficient to remove noise and isolate affected memory locations. However there were considerations taken into account that should this be insufficient to isolate the memory locations we would be able to perform more. As the iteration 3 of the tool was used in these experiments, snapshots were taken at user controlled intervals by interacting with the tool's command line interface.

For example, when testing the first location listed above, we defined the test as ″click 1 1″ in the tool to communicate that we were testing clicking on the x=1, y=1 location as shown in Figure 3.9. We then performed our user action of flagging or unflagging the tile and pressed enter on the tool to save the memory dump. As memory dumps are encoded with both the experiment run and snapshot number there was no confusion or mixing of experiment results during analysis. For example the tool can parse the filename ″Dump_click_1_1_1_4″ and know that it the result of the 4th snapshot of the 1st experiment run testing the user action ″click 1 1″.

The tool then analysed the snapshots and returned any memory locations it believes are connected to the user action performed. Finally, these results were verified by opening the process in Ollydbg and manually analysing the values of the locations returned by the memory tool.



```
Address  Hex dump
01005300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01005310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01005320 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01005330 0A 00 00 00 09 00 00 00 09 00 00 00 00 00 00 00
01005340 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005360 10 8E 41 40 41 8F 8F 41 40 40 10 0F 0F 0F 0F 0F
01005370 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005380 10 0F 42 40 41 42 42 41 40 40 10 0F 0F 0F 0F 0F
01005390 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
010053A0 10 8F 41 40 40 40 41 41 41 40 10 0F 0F 0F 0F 0F
010053B0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
010053C0 10 0F 41 40 40 40 41 8F 41 40 10 0F 0F 0F 0F 0F
010053D0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
010053E0 10 0F 41 41 41 40 41 41 41 40 10 0F 0F 0F 0F 0F
010053F0 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005400 10 0F 0F 8F 41 40 40 41 41 41 10 0F 0F 0F 0F 0F
01005410 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005420 10 0F 0F 0F 42 41 41 42 8F 0F 10 0F 0F 0F 0F 0F
01005430 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005440 10 8F 0F 0F 0F 8F 0F 0F 8F 0F 10 0F 0F 0F 0F 0F
01005450 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005460 10 0F 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F
01005470 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
01005480 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005490 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
```

Figure 4.1: Example result of verifying experiment run in Ollydbg. The tile being tested is at location 1,1 (top left corner, small red square outline). The game area in memory dump is shown by the large red square outline

4.2.3 Results

With each experiment run, it was found that ten snapshots were sufficient in isolating the key memory locations relevant to the user action of flagging a square. Furthermore it was found that only one memory location in each test was isolated and that memory location's value consistently alternated between 8E (flag) and 8F (no flag). The result of observing one such key location in Ollydbg for verification is shown in Figure 4.1.

The experiment demonstrated that each tile has a distinct location in memory containing a value for each state the tile could be in. Examples of such values and their meanings are listed below:

- 0F corresponds to an empty, unrevealed tile
- 8F corresponds to a mine
- 8E corresponds to a flagged tile with a mine under it

Furthermore, by locating multiple tiles, we were able to determine that there is a 2D array structure stored in memory representing the game board. Each tile on a given row of the board is stored adjacently in 4 bytes of memory, with some unknown bytes separating the rows. When viewed in Ollydbg, a clear 2D array can be seen which corresponds to the game board.

For each memory location the results were consistent. Restarting the Minesweeper game or using a 'fresh' versus modified board produced no difference in results. Furthermore, while the values of memory locations changed consistently, their addresses did not. For example, the "x=1, y=1" tile always had an address of 0x01005361, "x=1, y=2" always had an address of 0x01005362 (1 more than the "x=1, y=1"), "x=1, y=4" always had an address of 0x01005364 (3 more than the "x=1, y=1"), etc. This confirmed our assumption that memory locations would be fixed.

By combining these findings, we can therefore accomplish our goal of locating mines before they are revealed by obtaining the location in memory where the game board is stored and then finding any values of 9F in that region. By cross referencing the location of each 9F value in memory, we can then accurately determine where a mine is.

4.2.4 Conclusion

Through our testing on Minesweeper, it was found that the memory tool was useful in assisting with reverse engineering as we were successfully able to identify the data structure of the game and understand how parts of the game worked. As it was able to isolate the key memory location affected by the user action, we were able to locate and explore the game board in memory. Finally by using our information gained, we were able to identify hidden mines and avoid them in the game. This shows that tool is viable for strategically reducing noise and identifying useful memory locations to help us reverse engineer programs.

4.3 Notepad++

Notepad++ is a free text and source code editor for Windows released in 2003. It features basic editing and formatting options, resulting in a minimalistic operational footprint on any user's system. Basic formatting such as font and text size can be changed in this program through selecting text and clicking the format tab, but this program is not commonly used by those who wish to perform complex or extensive text formatting. Users can perform a variety of file editing tasks such as:

- Text input. Keyboard text input is inserted at the cursor position.
- Changing the position of the text input cursor by either clicking a position in the text and setting the cursor to that position, or using the arrow keys to move the cursor to a new position relative from its previous one.
- Selecting (highlighting) a region of text to perform further actions such as copy or paste on.

4.3.1 Goals of the analysis

The main aim of the analysis was to understand how Notepad++ handles its text input. In order to do this, we hoped to obtain the following information:

- Determine the data structure used to store text.
- Locate where the cursor position is stored in memory.
- Determine how a selected region of text is tracked in memory.

Often text editors make use of various data structures to optimise performance during text editing tasks such as:

- A Gap Buffer [28] is a dynamic buffer following the input cursor which is maintained by the text editor. When the user fills up the buffer by typing text or moves the cursor, the text editor creates a new buffer behind the input cursor by moving all text after the cursor backwards in memory. This allows the editor to efficiently implement insertion and deletion from the middle of a text block without having to rearrange the text in memory every time a user performs these actions.

- A Rope is a binary tree where leaves store individual strings and their lengths while nodes hold the sum of leaf lengths in its left subtree [13]. As a result, each node splits the overall string in two, with the left subtree holding the first part and right subtree holding the later half. Input is handled by splitting the existing tree at the location of the cursor, adding the new string to one of the subtrees then concatenating the resulting subtrees.

- A Piece Table is a data structure that tracks all insertions and deletions made to a document relative to its original version [20]. As both the original text and subsequent edits are saved in immutable blocks, no information is lost through editing. This allows the text editor to easily implement the undo functionality.

It was hoped that by analysing Notepad++'s memory changes we can discover which, if any, of these data structures are used in Notepad++.

4.3.2 Method

To perform the experiment, we used the latest version of the memory tool for analysis, Notepad++ 7.3 32 bit as the test target and Ollydbg for verification. As with Minesweeper, tests would be run on Notepad++ in a controlled environment and the memory tool tried to form relations between user actions and memory locations. Similar to the Minesweeper experiments, each set of Notepad++ actions were performed five times for a total of ten snapshots per experiment run. Finally, these memory locations would be viewed using the Ollydbg process memory viewer to determine if the memory tool made correct conclusions.

We input a controlled sequence of characters, such as "zxcvbnm", to investigate text input handing. This string was chosen initially as it is uncommon and thus easier to identify. To test the text input handling, the memory tool was run while this string was typed into an empty document in the following order:

- Snapshot taken with empty document
- Enter string "zxcvbnm" into document
- Snapshot taken with document now containing text
- Delete string "zxcvbnm" from document

This was also repeated with inserting a substring into the middle of a longer string.

We used a similar process to investigate cursor position and text selection. For cursor position we started off with the document already containing text, manipulated the cursor position and then compared snapshots. The text, initial cursor position and amount of cursor movement stayed constant between experiment runs. The process performed was:

- Snapshot taken with cursor position at initial place on page (first line, before first character)
- Cursor is moved to a defined location (second line, after 18th character)
- Snapshot taken with new cursor position
- Cursor moved back to original position on page (first line, before first character)

Finally, the text selection experiment was performed by comparing snapshots with and without a selected section of text. The text document and region of selection remained constant between experiments. The procedure involved:

- Snapshot taken with no text highlighted
- First 20 characters are highlighted
- Snapshot taken with text highlighted
- Text highlighting removed

4.3.3 Results

The memory tool was able to positively identify three memory regions that consistently changed as a result of simple user text input. When viewing these locations in Ollydbg, there was clear evidence that the text input was being saved plainly in memory, with each character of the string visible in memory. Furthermore, when inserting shorter strings (such as less than 5 letters), the same number of consistently changing memory regions were detected. However when inserting longer strings (such as more than 20 letters), the denoising tool failed to detect any consistently changing regions. The length of substring needed to fail the experiment changed between experiment runs. This suggests the possible presence of a gap buffer as longer insertions cause the total string length to exceed the capacity of the gap buffer and as a result the program moves the entire string to a larger buffer elsewhere in memory. Because the string now resides at a new memory location, the denoising tool does not detect it as consistent change.

Furthermore, 2 different memory locations were found to have changed as a result of cursor movement. By locating these 2 memory locations, we are able to perform further analysis and understand how they work. For example, when testing the cursor position of second line, 18th character, it was found that the following memory locations changed consistently:

- 0x03039ED8
- 0x03039EE0

While these results by themselves are not very meaningful, further investigation into these locations with Ollydbg presents the result shown in Figure 4.2. When the cursor was at position second line, 18th character, the values of the key memory locations were:

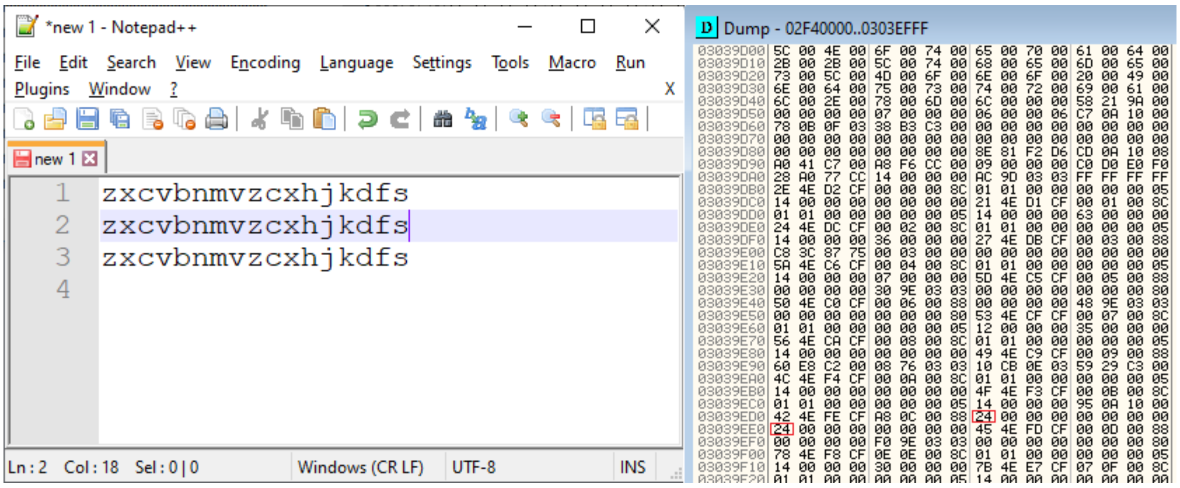
- 0x03039ED8 = 0x24
- 0x03039EE0 = 0x24

As a result, we can conclude that these memory locations are indeed of interest because the value 0x24 corresponds to the decimal value of 36. The cursor is at the 36th character in the document, hence we can deduce that Notepad++ stores cursor information as the character number it is located after.

Unfortunately, when analysing snapshots during the text highlighting experiment, the memory tool was unable to find any memory locations that changed consistently. It is possible that highlighting may be defined as a attribute of a text location rather than being tracked by a separate variable defining the start and end locations of a highlighted block of text.

4.3.4 Conclusion

Overall the memory tool was useful in locating memory regions related to user actions on Notepad++ and successfully assisted in analysis of two out of the three scenarios. It was able to remove noise from the experiment data and pinpoint the relevant memory location


*new 1 - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run
Plugins Window ?
new 1
Dump - 02F40000..0303EFFF

```
zxcvbnmvzcxhjkdfs
zxcvbnmvzcxhjkdfs
zxcvbnmvzcxhjkdfs

Ln : 2    Col : 18    Sel : 0 | 0        Windows (CR LF)    UTF-8        INS
```

```
03039D00 5C 00 4E 00 6F 00 74 00 65 00 70 00 61 00 64 00
03039D10 2B 00 2B 00 5C 00 74 00 68 00 65 00 6D 00 65 00
03039D20 73 00 5C 00 4D 00 6F 00 6E 00 6F 00 20 00 49 00
03039D30 6E 00 64 00 75 00 73 00 74 00 72 00 69 00 61 00
03039D40 6C 00 2E 00 78 00 6D 00 6C 00 00 00 58 21 9A 00
03039D50 00 00 00 00 07 00 00 00 06 00 00 06 C7 0A 10 00
03039D60 78 0B 0F 03 38 B3 C3 00 00 00 00 00 00 00 00 00
03039D70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
03039D80 00 00 00 00 00 00 00 00 8E 81 F2 D6 CD 0A 10 08
03039D90 A0 41 C7 00 A8 F6 CC 00 09 00 00 00 C0 D0 E0 F0
03039DA0 28 A0 77 CC 14 00 00 00 AC 9D 03 03 FF FF FF FF
03039DB0 2E 4E D2 CF 00 00 00 8C 01 01 00 00 00 00 00 05
03039DC0 14 00 00 00 00 00 00 00 21 4E D1 CF 00 01 00 8C
03039DD0 01 01 00 00 00 00 00 05 14 00 00 00 63 00 00 00
03039DE0 24 4E DC CF 00 02 00 8C 01 01 00 00 00 00 00 05
03039DF0 14 00 00 00 36 00 00 00 27 4E DB CF 00 03 00 88
03039E00 C8 3C 87 75 00 03 00 00 00 00 00 00 00 00 00 00
03039E10 5A 4E C6 CF 00 04 00 8C 01 01 00 00 00 00 00 05
03039E20 14 00 00 00 07 00 00 00 5D 4E C5 CF 00 05 00 88
03039E30 00 00 00 00 30 9E 03 03 00 00 00 00 00 00 00 80
03039E40 50 4E C0 CF 00 06 00 88 00 00 00 00 48 9E 03 03
03039E50 00 00 00 00 00 00 00 80 53 4E CF CF 00 07 00 8C
03039E60 01 01 00 00 00 00 00 05 12 00 00 00 35 00 00 00
03039E70 56 4E CA CF 00 08 00 8C 01 01 00 00 00 00 00 05
03039E80 14 00 00 00 00 00 00 00 49 4E C9 CF 00 09 00 88
03039E90 60 E8 C2 00 08 76 03 03 10 CB 0E 03 59 29 C3 00
03039EA0 4C 4E F4 CF 00 0A 00 8C 01 01 00 00 00 00 00 05
03039EB0 14 00 00 00 00 00 00 00 4F 4E F3 CF 00 0B 00 8C
03039EC0 01 01 00 00 00 00 00 05 14 00 00 00 95 0A 10 00
03039ED0 42 4E FE CF A8 0C 00 88 24 00 00 00 00 00 00 00
03039EE0 24 00 00 00 00 00 00 00 45 4E FD CF 00 0D 00 88
03039EF0 00 00 00 00 F0 9E 03 03 00 00 00 00 00 00 00 80
03039F00 78 4E F8 CF 0E 0E 00 8C 01 01 00 00 00 00 00 05
03039F10 14 00 00 00 30 00 00 00 7B 4E E7 CF 07 0F 00 8C
03039F20 01 01 00 00 00 00 00 05 14 00 00 00 00 00 00 00
```

Figure 4.2: Example result of verifying experiment run in Ollydbg. The cursor is at location second line, after 18th character in Notepad++. The consistently changing memory locations identified by the memory tool are shown in the red boxes in Ollydbg

for text input, thus allowing us to analyse and understand how Notepad++ handles its text. When using Ollydbg to verify the memory locations found by our tool, there was discernible change such as input text string actually being visible in the memory space. This shows that the memory locations discovered were correctly connected to the user action. When dealing with cursor locations, it was able to discover the memory locations containing location of the cursor and help us to understand how Notepad++ parses its cursor value. However, usability suffered in the third case (text selection) due to it requiring more information than just the application memory. A useful improvement might be a way to relate function calls to user actions similar to programs such as IDA Pro when the user is faced with more advanced problems that our current memory tool cannot solve.

4.4 Solitaire

Microsoft Solitaire is a single player game based off the classical card game solitaire, shipped with the Windows Operating System since version 3.0 in 1990 [5]. The game starts by creating seven columns of face down cards, with an increasing number of cards per column, starting with one card in the left column and ending with seven in the rightmost one. Only the first card of each column is revealed and the value of all others, including 24 cards in the 'deck' column, are hidden from the player. The win condition of the game is to accumulate all cards, sorted by suit, in ascending order in four 'foundation' columns above the playing area. Players may drag cards to move them between different columns subject to certain restrictions such as:

- A card may be moved to another column in the playing area only if the card above it is exactly one higher and of a different color. For example, a Queen of Spades may be moved below a King of Hearts, but not a King of Clubs (same color) nor a Ace of Hearts (Queen is not one lower than an Ace).

- When a card is moved, all cards below it must also be moved.

- Only one card of the deck column may be revealed at once.

The objective of the game is to correctly sort all the cards into the four foundation columns in the least number of moves. While game is lost when no legal moves are possibles, the Microsoft version of solitaire allows the player to undo moves, thus making losses less likely. As it was originally designed to help familiarise Windows users with a mouse [5], all actions in the game involve moving cards by using the mouse to either click on the card and then clicking its target destination or simply dragging the card to its destination.

4.4.1 Goals of the analysis

The aim of the experiment was to determine whether or not it is possible for a person to use the memory tool and gain an advantage in the solitaire game by seeing cards before they are revealed. This is desirable because knowing cards before they are revealed allows the player to formulate strategies around future cards and have a higher chance of winning or getting a better score. In order to achieve this, we aim to obtain the following information:

- Determine memory layout for cards on playing field.
- Determine memory layout for cards on deck and foundation pile.
- Determine unrevealed cards.

For card games of this type, typical data structures used to store cards include:

- Simple arrays for card stacks on the playing field. More than one card may be revealed at once in a pile and arrays provide access to each element in order to allow this gameplay.
- Linked lists for the deck and foundation pile. As only one card at a time is accessible in either pile, a linked list may be used to facilitate the player accessing the top card. Furthermore, because the cards in the deck are saved in a set order upon game creation, a linked list would efficiently preserve this order.

4.4.2 Method

To perform this experiment, we ran the Microsoft Solitaire game alongside our memory tool and used Ollydbg to verify results. The Windows XP version was used again as Microsoft no longer includes Solitaire with newer version of Windows. We started off with a fresh game of solitaire that had an available move on the game board and then performed the following actions to gather results:

- Snapshot the current game state
- Perform a legal action by moving a card (to either another pile on the playing field or foundation pile)
- Snapshot game state after the move
- Use undo function to reset game to initial state

The undo button was important as it allowed us to reset the game to the exact state at the start of game without randomising any cards that a 'new game' would cause.

Similarly, we were able to analyse unrevealed cards by playing the game until we were in a game state with a revealable card as shown in Figure 4.3. Then we took snapshots using the following process:

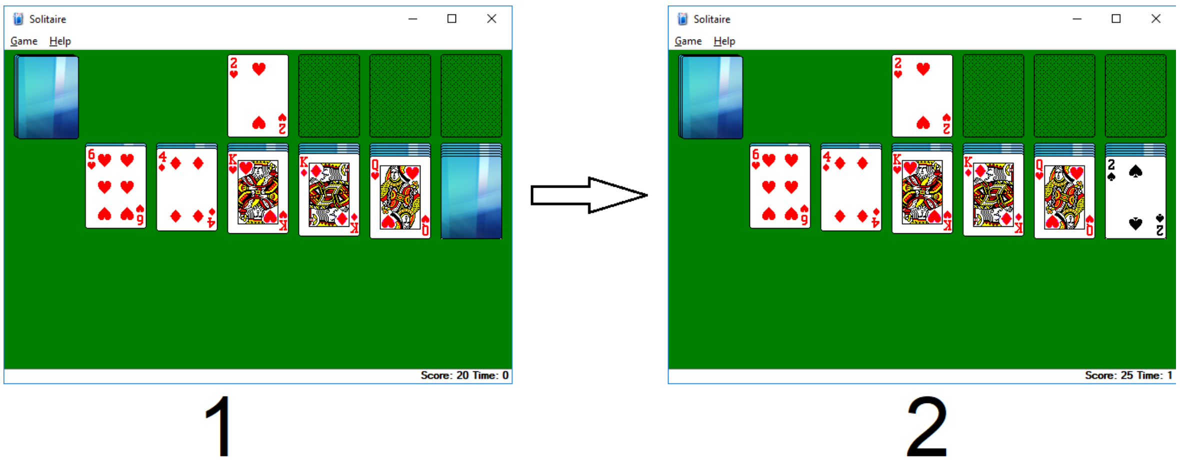


Figure 4.3: Game states used to analyse unrevealed cards. State 1 contains a revealable card (rightmost column), State 2 shows the game after this card is revealed

- Use memory tool to take a snapshot of the current game state (State 1 in Figure 4.3)
- Reveal the card
- Use memory tool to take a snapshot of the new game state (State 2 in Figure 4.3)
- Use undo function to reset game to initial state

While each action (such as moving a card from pile 1 to 2 on the game board) was dependant upon not only the action itself but also the game state at the time of the action (see Figure 4.4), the procedure largely remained the same as previous experiments. Each set of actions were repeated 5 times for a total of ten snapshots before the memory tool analysed them and attempted to find memory locations related to the user action.

4.4.3 Results

It was found that, similar to Minesweeper, the cards in both the foundation and playing field columns were arranged sequentially with some delimiting data between each card. When a card is revealed, 8 bytes are changed consistently in memory, corresponding to the location of the card. For example, revealing a card in the rightmost stack always causes 8 bytes of memory starting at 0x100719C to be set to 0x00 as shown in Figures 4.5, 4.6 and 4.7. This location was consistently identified by the memory tool across multiple experiment runs, and did not change even after restarting the game.

This reveals information about the memory structure of Solitaire. First of all it appears that the game always keeps track of the 'next unrevealed' card in a pile with a separate array data structure that contains the cards in play. As shown in Figures 4.5, 4.6 and 4.7, the memory space goes through a number of changes as the game is played :

- Card is unrevealed. 0x100719C - 0x10071A3 is populated with the data of a card (Two of Hearts)
- Card is revealed. 0x100719C - 0x10071A3 is cleared of data
- Two of Hearts is played, resulting in a new unrevealed card present at the top of the pile. 0x100719C - 0x10071A3 is populated with the data of a new card (Five of Diamonds)

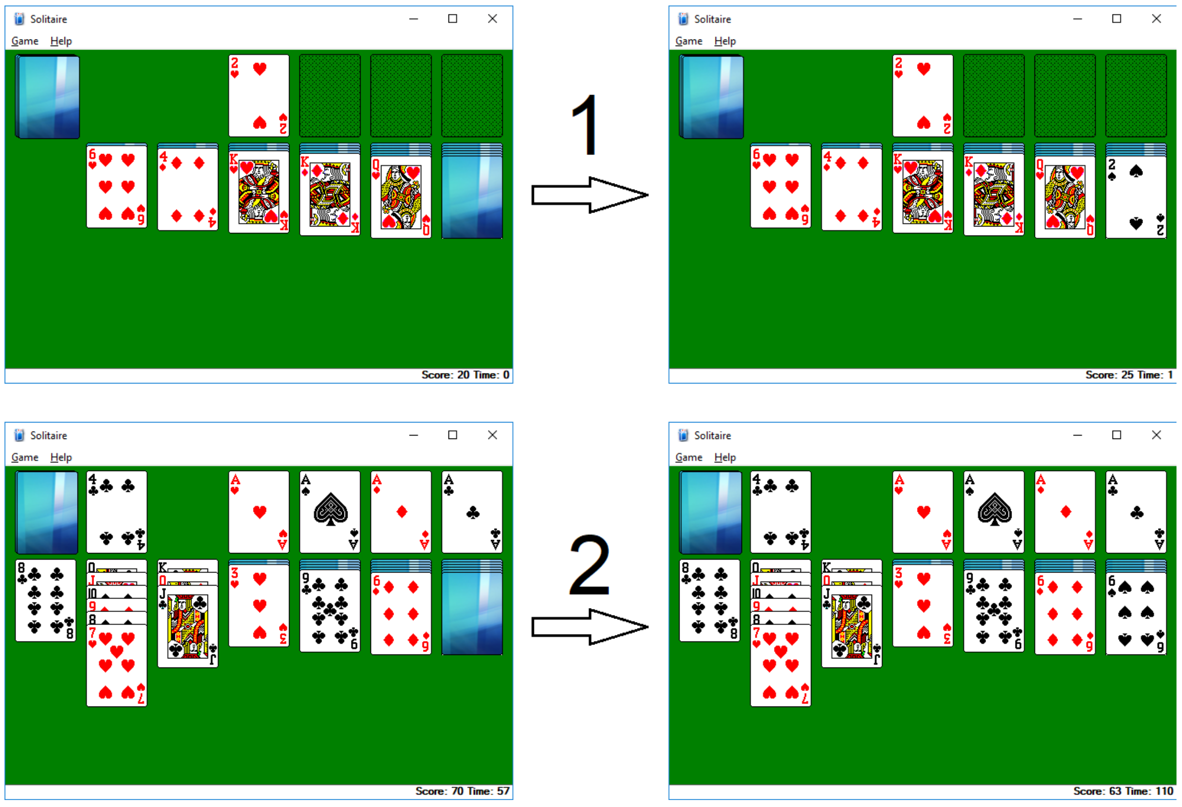
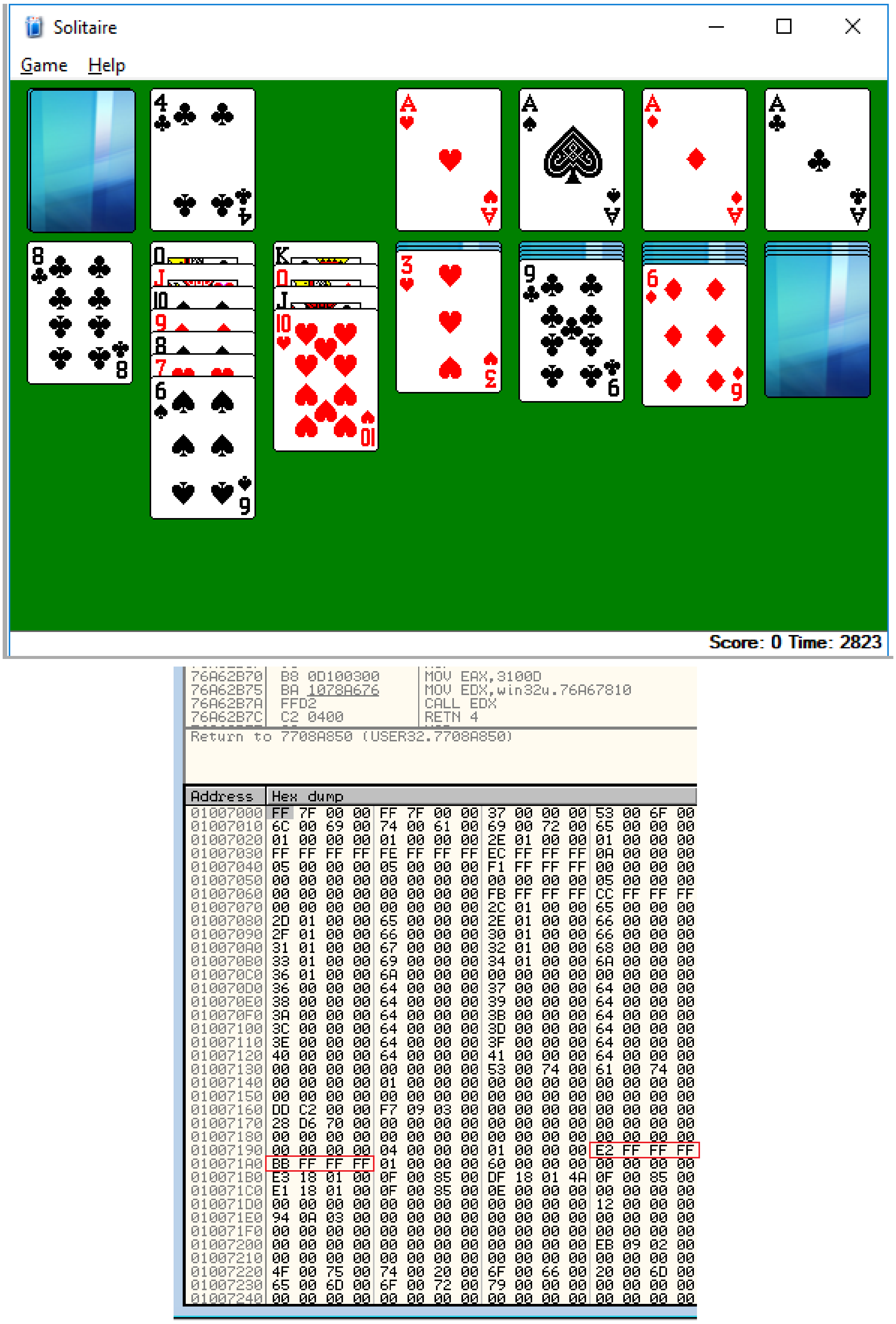


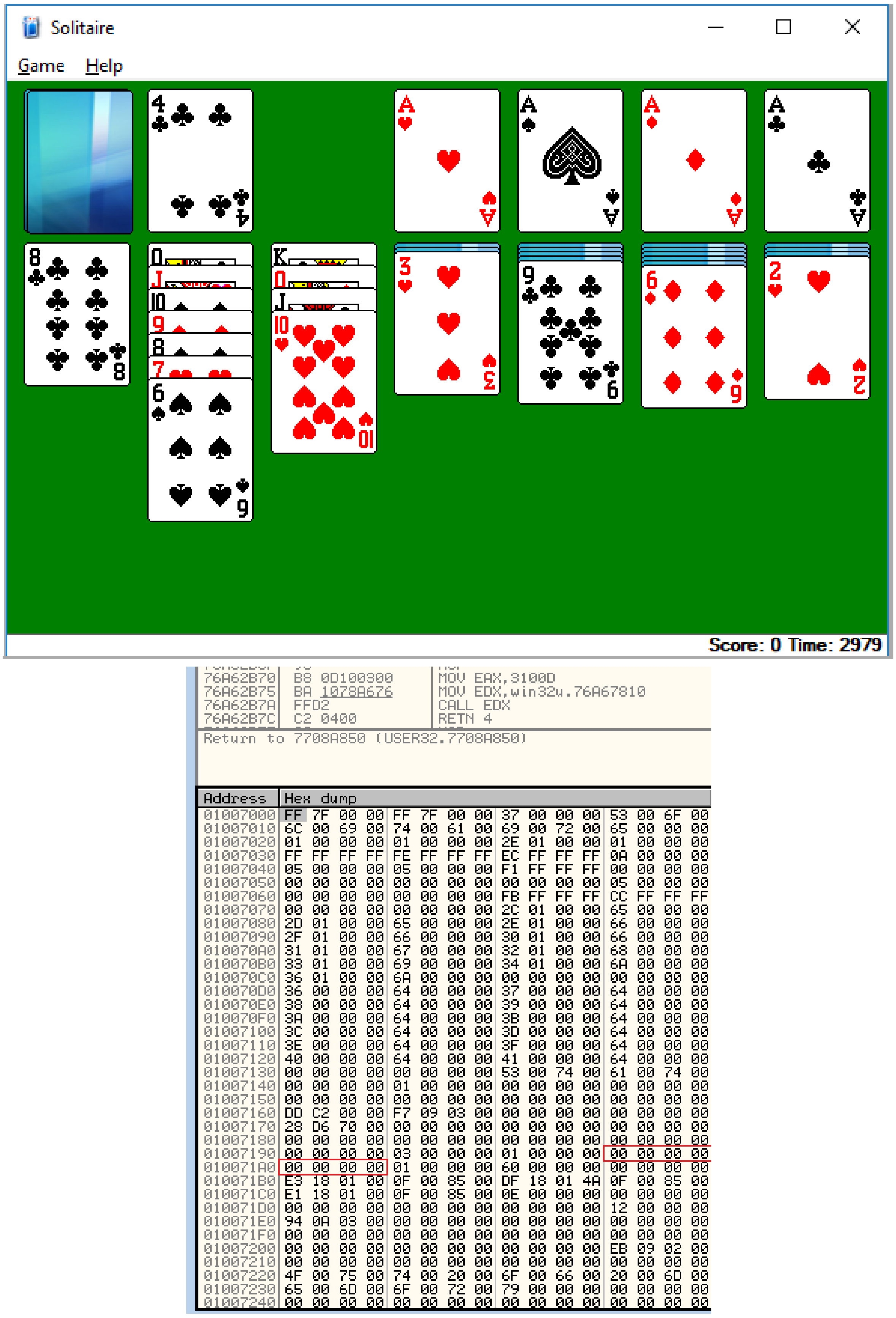
Figure 4.4: In both experiments, the action being tested is revealing the 6th card on the 7th column. However, the same user action under different initial game states (compare game 1 versus game 2) will produce different changes in game state and memory



```
76A62B70  B8 0D100300    MOV EAX,3100D
76A62B75  BA 1078A676    MOV EDX,win32u.76A67810
76A62B7A  FFD2           CALL EDX
76A62B7C  C2 0400        RETN 4
Return to 7708A850 (USER32.7708A850)

Address   Hex dump
01007000  FF 7F 00 00 FF 7F 00 00 37 00 00 00 53 00 6F 00
01007010  6C 00 69 00 74 00 61 00 69 00 72 00 65 00 00 00
01007020  01 00 00 00 01 00 00 00 2E 01 00 00 01 00 00 00
01007030  FF FF FF FF FE FF FF FF EC FF FF FF 0A 00 00 00
01007040  05 00 00 00 05 00 00 00 F1 FF FF FF 00 00 00 00
01007050  00 00 00 00 00 00 00 00 00 00 00 00 05 00 00 00
01007060  00 00 00 00 00 00 00 00 FB FF FF FF CC FF FF FF
01007070  00 00 00 00 00 00 00 00 2C 01 00 00 65 00 00 00
01007080  2D 01 00 00 65 00 00 00 2E 01 00 00 66 00 00 00
01007090  2F 01 00 00 66 00 00 00 30 01 00 00 66 00 00 00
010070A0  31 01 00 00 67 00 00 00 32 01 00 00 68 00 00 00
010070B0  33 01 00 00 69 00 00 00 34 01 00 00 6A 00 00 00
010070C0  36 01 00 00 6A 00 00 00 00 00 00 00 00 00 00 00
010070D0  36 00 00 00 64 00 00 00 37 00 00 00 64 00 00 00
010070E0  38 00 00 00 64 00 00 00 39 00 00 00 64 00 00 00
010070F0  3A 00 00 00 64 00 00 00 3B 00 00 00 64 00 00 00
01007100  3C 00 00 00 64 00 00 00 3D 00 00 00 64 00 00 00
01007110  3E 00 00 00 64 00 00 00 3F 00 00 00 64 00 00 00
01007120  40 00 00 00 64 00 00 00 41 00 00 00 64 00 00 00
01007130  00 00 00 00 00 00 00 00 53 00 74 00 61 00 74 00
01007140  00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
01007150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007160  DD C2 00 00 F7 09 03 00 00 00 00 00 00 00 00 00
01007170  28 D6 70 00 00 00 00 00 00 00 00 00 00 00 00 00
01007180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007190  00 00 00 00 04 00 00 00 01 00 00 00 E2 FF FF FF
010071A0  BB FF FF FF 01 00 00 00 60 00 00 00 00 00 00 00
010071B0  E3 18 01 00 0F 00 85 00 DF 18 01 4A 0F 00 85 00
010071C0  E1 18 01 00 0F 00 85 00 0E 00 00 00 00 00 00 00
010071D0  00 00 00 00 00 00 00 00 00 00 00 00 12 00 00 00
010071E0  94 0A 03 00 00 00 00 00 00 00 00 00 00 00 00 00
010071F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007200  00 00 00 00 00 00 00 00 00 00 00 00 EB 09 02 00
01007210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007220  4F 00 75 00 74 00 20 00 6F 00 66 00 20 00 6D 00
01007230  65 00 6D 00 6F 00 72 00 79 00 00 00 00 00 00 00
01007240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

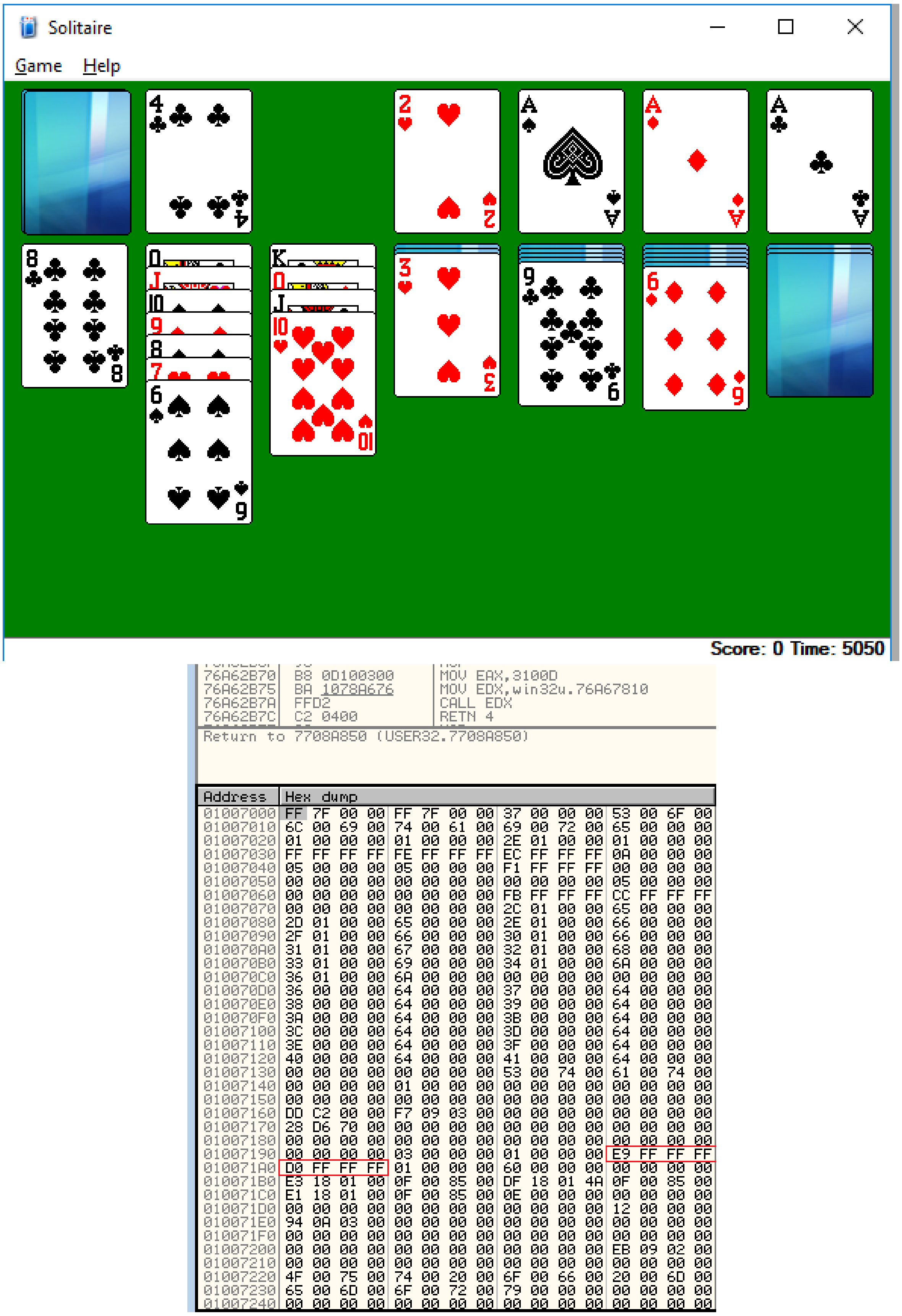
Figure 4.5: Diagram showing changes in memory data and game state. This is the state before revealing the card on the rightmost column. The relevant memory location is highlighted in red



```
76A62B70  B8 0D100300      MOV EAX,3100D
76A62B75  BA 1078A676      MOV EDX,win32u.76A67810
76A62B7A  FFD2             CALL EDX
76A62B7C  C2 0400          RETN 4
Return to 7708A850 (USER32.7708A850)

Address   Hex dump
01007000  FF 7F 00 00 FF 7F 00 00 37 00 00 00 53 00 6F 00
01007010  6C 00 69 00 74 00 61 00 69 00 72 00 65 00 00 00
01007020  01 00 00 00 01 00 00 00 2E 01 00 00 01 00 00 00
01007030  FF FF FF FF FE FF FF FF EC FF FF FF 0A 00 00 00
01007040  05 00 00 00 05 00 00 00 F1 FF FF FF 00 00 00 00
01007050  00 00 00 00 00 00 00 00 00 00 00 00 05 00 00 00
01007060  00 00 00 00 00 00 00 00 FB FF FF FF CC FF FF FF
01007070  00 00 00 00 00 00 00 00 2C 01 00 00 65 00 00 00
01007080  2D 01 00 00 65 00 00 00 2E 01 00 00 66 00 00 00
01007090  2F 01 00 00 66 00 00 00 30 01 00 00 66 00 00 00
010070A0  31 01 00 00 67 00 00 00 32 01 00 00 68 00 00 00
010070B0  33 01 00 00 69 00 00 00 34 01 00 00 6A 00 00 00
010070C0  36 01 00 00 6A 00 00 00 00 00 00 00 00 00 00 00
010070D0  36 00 00 00 64 00 00 00 37 00 00 00 64 00 00 00
010070E0  38 00 00 00 64 00 00 00 39 00 00 00 64 00 00 00
010070F0  3A 00 00 00 64 00 00 00 3B 00 00 00 64 00 00 00
01007100  3C 00 00 00 64 00 00 00 3D 00 00 00 64 00 00 00
01007110  3E 00 00 00 64 00 00 00 3F 00 00 00 64 00 00 00
01007120  40 00 00 00 64 00 00 00 41 00 00 00 64 00 00 00
01007130  00 00 00 00 00 00 00 00 53 00 74 00 61 00 74 00
01007140  00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
01007150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007160  DD C2 00 00 F7 09 03 00 00 00 00 00 00 00 00 00
01007170  28 D6 70 00 00 00 00 00 00 00 00 00 00 00 00 00
01007180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007190  00 00 00 00 03 00 00 00 01 00 00 00 00 00 00 00
010071A0  00 00 00 00 01 00 00 00 60 00 00 00 00 00 00 00
010071B0  E3 18 01 00 0F 00 85 00 DF 18 01 4A 0F 00 85 00
010071C0  E1 18 01 00 0F 00 85 00 0E 00 00 00 00 00 00 00
010071D0  00 00 00 00 00 00 00 00 00 00 00 00 12 00 00 00
010071E0  94 0A 03 00 00 00 00 00 00 00 00 00 00 00 00 00
010071F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007200  00 00 00 00 00 00 00 00 00 00 00 00 EB 09 02 00
01007210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007220  4F 00 75 00 74 00 20 00 6F 00 66 00 20 00 6D 00
01007230  65 00 6D 00 6F 00 72 00 79 00 00 00 00 00 00 00
01007240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4.6: Diagram showing changes in memory data and game state. This is the state after revealing the card on the rightmost column



```
76A62B70 B8 0D100300   MOV EAX,3100D
76A62B75 BA 1078A676   MOV EDX,win32u.76A67810
76A62B7A FFD2          CALL EDX
76A62B7C C2 0400       RETN 4
Return to 7708A850 (USER32.7708A850)

Address  Hex dump
01007000 FF 7F 00 00 FF 7F 00 00 37 00 00 00 53 00 6F 00
01007010 6C 00 69 00 74 00 61 00 69 00 72 00 65 00 00 00
01007020 01 00 00 00 01 00 00 00 2E 01 00 00 01 00 00 00
01007030 FF FF FF FF FE FF FF FF EC FF FF FF 0A 00 00 00
01007040 05 00 00 00 05 00 00 00 F1 FF FF FF 00 00 00 00
01007050 00 00 00 00 00 00 00 00 00 00 00 00 05 00 00 00
01007060 00 00 00 00 00 00 00 00 FB FF FF FF CC FF FF FF
01007070 00 00 00 00 00 00 00 00 2C 01 00 00 65 00 00 00
01007080 2D 01 00 00 65 00 00 00 2E 01 00 00 66 00 00 00
01007090 2F 01 00 00 66 00 00 00 30 01 00 00 66 00 00 00
010070A0 31 01 00 00 67 00 00 00 32 01 00 00 68 00 00 00
010070B0 33 01 00 00 69 00 00 00 34 01 00 00 6A 00 00 00
010070C0 36 01 00 00 6A 00 00 00 00 00 00 00 00 00 00 00
010070D0 36 00 00 00 64 00 00 00 37 00 00 00 64 00 00 00
010070E0 38 00 00 00 64 00 00 00 39 00 00 00 64 00 00 00
010070F0 3A 00 00 00 64 00 00 00 3B 00 00 00 64 00 00 00
01007100 3C 00 00 00 64 00 00 00 3D 00 00 00 64 00 00 00
01007110 3E 00 00 00 64 00 00 00 3F 00 00 00 64 00 00 00
01007120 40 00 00 00 64 00 00 00 41 00 00 00 64 00 00 00
01007130 00 00 00 00 00 00 00 00 53 00 74 00 61 00 74 00
01007140 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
01007150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007160 DD C2 00 00 F7 09 03 00 00 00 00 00 00 00 00 00
01007170 28 D6 70 00 00 00 00 00 00 00 00 00 00 00 00 00
01007180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007190 00 00 00 00 03 00 00 00 01 00 00 00 E9 FF FF FF
010071A0 D0 FF FF FF 01 00 00 00 60 00 00 00 00 00 00 00
010071B0 E3 18 01 00 0F 00 85 00 DF 18 01 4A 0F 00 85 00
010071C0 E1 18 01 00 0F 00 85 00 0E 00 00 00 00 00 00 00
010071D0 00 00 00 00 00 00 00 00 00 00 00 00 12 00 00 00
010071E0 94 0A 03 00 00 00 00 00 00 00 00 00 00 00 00 00
010071F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007200 00 00 00 00 00 00 00 00 00 00 00 00 EB 09 02 00
01007210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01007220 4F 00 75 00 74 00 20 00 6F 00 66 00 20 00 6D 00
01007230 65 00 6D 00 6F 00 72 00 79 00 00 00 00 00 00 00
01007240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4.7: Diagram showing changes in memory data and game state. This is the state after moving the Two of Hearts, resulting in a different card being at the top of the rightmost column. This new card's memory data is highlighted in red

The contents of the key memory location are consistently different depending on what card is unrevealed. Around 20 games of Solitaire later, another game arose where the Two of Hearts was the next unrevealed card in the rightmost pile. In this game, memory locations 0x100719C - 0x10071A3 contained the same data as shown above: the values "E2 FF FF FF BB FF FF FF". This showed that we are able to see cards before they are played because each pile has a single memory region for its unrevealed card, and there is a one-to-one relation between the card that is unrevealed and the data stored in the memory location.

Unfortunately, we were not able to use the tool to locate and analyse changes to the foundation piles caused by the user action of moving a card from the playing area to the foundation pile. The tool did not locate any memory locations that changed consistently. This is possibly because unlike the static cards generated by the game in the playing field, the foundation piles use a dynamically allocated data structure such as a linked list to handle cards that have been placed there. As a result because the size of the data structure may expand over the course of the game via malloc operations, the memory tool was unable to find a consistently changing region of memory.

4.4.4 Conclusion

The tool was useful and helped to pinpoint the critical memory locations that enabled further analysis to understand the Solitaire memory structure. As a result we were able to see cards before they were revealed.

However, the memory tool did have a limitation as it did not fully support analysis on events that were dependant on an initial game state. As a result, while it correctly obtained memory information regarding what locations were affected by the user action of moving a card, there was no way to encode the prior game state into the snapshots being stored on disk. While this did not adversely affect the analysis as I could use my own knowledge of the game state (such as what card was moved and what cards were in play), it meant that we could not realistically perform analysis on these memory dumps at a later time if we needed to.

Chapter 5

Conclusion

The difficulty of reverse engineering has increased as modern software has become more sophisticated. To ensure reverse engineering is still possible, the development of new tools and techniques have been necessary. We aimed to simplify the reverse engineering process by removing noise from samples of program memory, thus leaving more useful data for the reverse engineer to examine. In order to achieve this goal, we created a tool that took snapshots of the target program's memory while a user interacted with it in a consistent way. Upon completion of a series of snapshots, the memory tool then performed an intersection operation to identify regions of consistent change which are hypothesized to be as a result of the user action. The user can then examine these regions of consistent change instead of parsing the entire memory space.

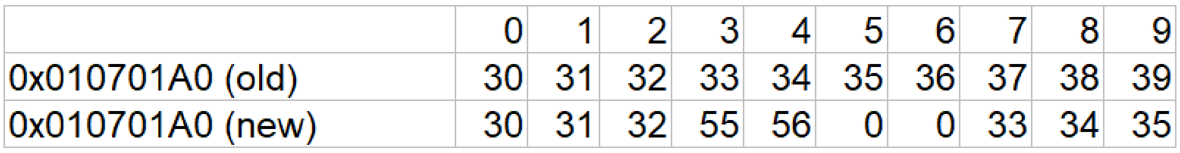
Overall, the memory tool has been helpful in assisting reverse engineering of applications. It was effective in quickly locating important sections of memory being affected by user actions and thus allows a reverse engineer to save time and effort when working with more complex applications. In all three cases studied, usage of the tool was important in achieving the goals of the analysis, but due to the prototype nature of the system it suffered from some limitations as well.

The most important benefit of using an automatic noise reduction tool is being able to rapidly find critical memory regions. On even small games such as Minesweeper or Solitaire, manually poring through megabytes of memory data is tedious and prone to human error. With the tool, such an operation takes mere minutes to complete and the number of memory locations that need to be manually assessed drops from megabytes to a mere few bytes. The removal of millions of bytes of noise is greatly useful to increasing human efficiency. Furthermore, the ability to curate experimental data for future use is important as it allows more advanced analysis by future researchers or even future versions of the tool without losing any original experiment data. By encoding the experiment conditions, such as user action performed, into the filename we reduce the chances of confusion or mix ups of experiment results

After three iterations, we reached a viable prototype that helps a reverse engineer analyse memory by removing noise from program memory snapshots. Using this tool, we were able to successfully obtain information from three Windows programs to illustrate the reverse engineering process that occurs in more complex scenarios.

5.1 Future Work

However, because this system is a prototype, it was unable to handle all forms of user action. We ran into issues using it in more complex cases such as trying to find highlighted



	0	1	2	3	4	5	6	7	8	9
0x010701A0 (old)	30	31	32	33	34	35	36	37	38	39
0x010701A0 (new)	30	31	32	55	56	0	0	33	34	35

Figure 5.1: An illustration of the changes in a program's memory space. In this example, the addition of a gap buffer has caused values from 33 and onwards to be moved four bytes to the right

text in Notepad++. As the program assumes that a user action directly causes a change in a fixed memory location, applications that do not adhere to this assumption cannot be easily analysed using the memory tool. In order to solve this issue, we may need to bring more advanced techniques into the tool and give it more intelligence or domain knowledge. An example of this would be the implementation of a better intersect function in the tool, and its effects on the results the tool produces when used on a program with a dynamically changing data structure as shown in Figure 5.1. In the current iteration of the tool, all memory values from 0x010701A3 and onwards would be reported as changed as a result of the user action. However in reality only 0x010701A3-0x010701A6 have been changed, and the rest of the unchanged data has simply been shifted 4 bytes. A possible fix for issues such as this would be the implementation of more advanced data comparison algorithms to solve the ”longest common subsequence” problem. By doing so, we would be able to define the longest common subsequences of memory data between two snapshots as the ”unchanged” region, thus allowing our system to even more effectively remove noise. With such an implementation, the results from the example in Figure 5.1 would change. Both the regions 0x0 - 0x2 as well as 0x7 onwards in the new data would be considered unchanged as they match memory sequences from the old data. Only 0x3-0x6 would be considered the change in memory, which is more likely to be what a user wants.

Bibliography

[1] The begining of the end for wrobot for official servers. https://wrobot.eu/articles/news/the-begining-of-the-end-of-wrobot-for-official-servers-r124/.

[2] Buffer overflows for dummies. https://www.sans.org/reading-room/whitepapers/threats/buffer-overflows-dummies-481.

[3] *Dynamic Binary Instrumentation for Deobfuscation and Unpacking*, IN-DEPTH SECURITY CONFERENCE 2009 EUROPE; IN-DEPTH SECURITY CONFERENCE 2009 EUROPE, Nov 2009, Vienne, Austria. 2009, HAL CCSD.

[4] How to dump wow from memory.... https://www.ownedcore.com/forums/world-of-warcraft/world-of-warcraft-bots-programs/wow-memory-editing/640683-how-dump-wow-memory.html.

[5] Interviews : Wes cherry. https://b3ta.com/interview/solitaire/.

[6] ow unpack - remove overwatch encryption statically. https://www.ownedcore.com/forums/fps/overwatch-exploits-hacks/553170-ow_unpack-remove-overwatch-encryption-statically.html.

[7] r/overwatch - [serious] how does blizzard plan on going about anti-cheat? https://www.reddit.com/r/Overwatch/comments/40ael4/serious_how_does_blizzard_plan_on_going_about/cytsmuj/.

[8] The sad news - honorbuddy and others. https://www.thebuddyforum.com/threads/the-sad-news-honorbuddy-and-others.411956/.

[9] Winmine. http://www.minesweeper.info/downloads/WinmineXP.html.

[10] World of warcraft honorbuddy ban wave on may, 2015. https://unbanservice.com/world-of-warcraft-honorbuddy-ban-wave-on-may-2015/.

[11] World of warcraft honorbuddy ban wave on may 2016. https://unbanster.com/wow-ban-wave-may-2016/.

[12] World of warcraft honorbuddy ban wave on november 2016. https://unbanster.com/wow-ban-wave-november-2016/.

[13] Ropes ccture (fast string concatenation). https://www.geeksforgeeks.org/ropes-data-structure-fast-string-concatenation/, Sep 2019.

[14] Alazab, M., Venkataraman, S., and Watters, P. Towards understanding malware behaviour by the extraction of api calls. *2010 Second Cybercrime and Trustworthy Computing Workshop* (2010), pp. 52–59.

[15] BACCI, A., BARTOLI, A., MARTINELLI, F., MEDVET, E., MERCALDO, F., AND VISAGGIO, C. A. Impact of code obfuscation on android malware detection based onstatic and dynamic analysis. *Proceedings of the 4th International Conference on Information Systems Security and Privacy* (2018), pp. 379385.

[16] BREW, K. Reverse engineering malware. https://www.alienvault.com/blogs/labs-research/reverse-engineering-malware.

[17] BROLL, D. W. Expert opinion on the influences of bots on the economy and gaming enjoyment in mmorpgs. Bossland GmbH (29 March 2012) . https://www.honorbuddy.com/Expert_opinion_bots.pdf.

[18] BRUENING, D., ZHAO, Q., AND AMARASINGHE, S. P. Transparent dynamic instrumentation. In *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012)* (2012), S. Hand and D. D. Silva, Eds., ACM, pp. 133–144.

[19] CHIKOFSKY, E. J., AND II, J. H. C. Reverse engineering and design recovery: A taxonomy. *IEEE Software 7*, 1 (1990), 13–17.

[20] CROWLEY, C. Data structures for text sequences. https://www.cs.unm.edu/~crowley/papers/sds.pdf, Sept. 30 1998.

[21] DAVID, R., BARDIN, S., TA, T. D., MOUNIER, L., FEIST, J., POTET, M., AND MARION, J. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016), vol. 1, pp. 653–656.

[22] DELGADO, T. T. Column: 'beyond tetris' - minesweeper. http://www.gamesetwatch.com/2007/02/column_beyond_tetris_minesweep.php.

[23] DERPY. Honorbuddy user manual. https://webcache.googleusercontent.com/search?q=cache:2yALdORFvwUJ:https://www.thebuddyforum.com/attachments/honorbuddy-user-manual-pdf.80490/+&cd=2&hl=en&ct=clnk&gl=nz.

[24] EENY. Getting started with wrobot. https://www.youtube.com/watch?v=UYgB_RjwV8s.

[25] EINAR. Why is disassembly not an exact science? https://reverseengineering.stackexchange.com/questions/2580/why-is-disassembly-not-an-exact-science.

[26] EMMERIK, M. V., AND WADDINGTON, T. Using a decompiler for real-world source recovery. In *Working Conference on Reverse Engineering* (2004), IEEE Computer Society, pp. 27–36.

[27] FEDERICO, A., PAYER, M., AND AGOSTA, G. rev.ng: a unified binary analysis framework to recover cfgs and function boundaries. pp. 131–141.

[28] *GNU Emacs Manual, Emacs Version 19*. Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA. Available on-line with the GNU Emacs distribution.

[29] HAWKINS, W., HISER, J. D., NGUYEN-TUONG, A., CO, M., AND DAVIDSON, J. W. Securing binary code. *IEEE Security Privacy 15*, 6 (2017), 77–81.

[30] HAWKINS, W., HISER, J. D., NGUYEN-TUONG, A., CO, M., AND DAVIDSON, J. W. Securing binary code. *IEEE Security Privacy 15*, 6 (2017), 77–81.

[31] HAWKINS, W. H., HISER, J. D., CO, M., NGUYEN-TUONG, A., AND DAVIDSON, J. W. Zipr: Efficient static binary rewriting for security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2017), pp. 559–566.

[32] INSTITUTE, A.-T. T. I. I.-S. Malware statistics and trends report. `https://www.av-test.org/en/statistics/malware/`.

[33] JBRAUMAN. How I handle objects. `https://www.ownedcore.com/forums/world-of-warcraft/world-of-warcraft-bots-programs/wow-memory-editing/208754-guide-kind-of-how-i-handle-objects.html`.

[34] JUNG, M., KIM, S., HAN, H., CHOI, J., AND CHA, S. K. B2r2: Building an efficient front-end for binary analysis.

[35] KRAMER, S., AND BRADFIELD, J. C. A general definition of malware. *Journal in Computer Virology* (2010), pp. 105–114.

[36] KUPERMAN, BRODLEY, OZDOGANOGLU, VIJAYKUMAR, AND JALOTE. Detection and prevention of stack buffer overflow attacks. *CACM: Communications of the ACM 48* (2005), pp. 50–56.

[37] LI, X., SHAN, Z., LIU, F., CHEN, Y., AND HOU, Y. A consistently-executing graph-based approach for malware packer identification. *IEEE Access 7* (2019), 51620–51629.

[38] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices 40*, 6 (June 2005), 190–200.

[39] MALANOV, A. Antivirus fundamentals: Viruses, signatures, disinfection. `https://www.kaspersky.com/blog/signature-virus-disinfection/13233/`.

[40] MCSHERRY, C. A new gaming feature: Spyware. `https://www.eff.org/deeplinks/2005/10/new-gaming-feature-spyware`, Oct 2011.

[41] MENG, X., AND MILLER, B. P. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, Association for Computing Machinery, p. 2435.

[42] MITTERHOFER, S., KRUEGEL, C., KIRDA, E., AND PLATZER, C. Server-side bot detection in massively multiplayer online games. *IEEE Security & Privacy 7*, 3 (May 2009), 29–36.

[43] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M. K., AND ZDANCEWIC, S. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009* (2009), M. Hind and A. Diwan, Eds., ACM, pp. 245–258.

[44] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy code. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, Jan. 16–18, 2002), pp. 128–139.

[45] O'Kane, P., Sezer, S., and McLaughlin, K. Obfuscation: The hidden malware. *IEEE Security & Privacy 9*, 5 (2011), 41–47.

[46] Olszewski, M., Ansel, J., and Amarasinghe, S. P. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (14th ASPLOS'09)* (Washington, DC, USA, Mar. 2009), ACM, pp. 97–108.

[47] Payne, M. F. Automating instrumentation: Identifying instrumentation points for monitoring constraints at runtime. Master's thesis, University of Texas at El Paso, 2014.

[48] Platzer, C. Sequence-based bot detection in massive multiplayer online games. In *International Conference on Information and Communications Security* (2011), IEEE, pp. 1–5.

[49] Polakowski, M., and Stewart, D. J. Warships of the first punic war: An archaeological investigation and contributory reconstruction of the egadi 10 warship from the battle of the egadi islands (241 b.c.). Master's thesis, East Carolina University, 2016.

[50] Raynal, F. Deobfuscation: recovering an ollvm-protected program. `https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html`.

[51] Reiss, S. P., and Renieris, M. Languages for dynamic instrumentation. `http://cs.brown.edu/~spr/research/bloom/dyninst.pdf`, Dec. 23 2003.

[52] Roccia, T. Malware packers use tricks to avoid analysis, detection. *McAfee Blogs Oct* (2018).

[53] Rosu, G., Schulte, W., and Serbanuta, T.-F. Runtime verification of c memory safety. In *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers* (2009), S. Bensalem and D. A. Peled, Eds., vol. 5779 of *Lecture Notes in Computer Science*, Springer, pp. 132–151.

[54] Russinovich, M., and Margosis, A. *Windows Sysinternals administrators reference*. Microsoft Press, 2012.

[55] Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C., and Bringas, P. G. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings* (2010), F. Massacci, D. S. Wallach, and N. Zannone, Eds., vol. 5965 of *Lecture Notes in Computer Science*, Springer, pp. 35–43.

[56] Scott, J. Detecting malware through static and dynamic techniques. `https://technical.nttsecurity.com/post/102efk4/detecting-malware-through-static-and-dynamic-techniques`.

[57] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. Report, Department of Computer Science, Stanford University, Stanford, CA, USA, Sept. 2007.

[58] Shahzad, R. K., Lavesson, N., and Johnson, H. Accurate adware detection using opcode sequence extraction. In *ARES* (2011), IEEE Computer Society, pp. 189–195.

[59] Sharif, M. I., Yegneswaran, V., Saïdi, H., Porras, P. A., and Lee, W. Eureka: A framework for enabling static malware analysis. In *ESORICS* (2008), S. Jajodia and J. López, Eds., vol. 5283 of *Lecture Notes in Computer Science*, Springer, pp. 481–500.

[60] SIMS, S. Introduction to reverse engineering for penetration testers. https://www.sans.org/summit-archives/file/summit-archive-1510603949.pdf.

[61] THOMAS MANDL, ULRICH BAYER, F. N. Anubis analyzing unknown binaries the automatic way. Virus Bulletin Conference 2009, Geneva, 2009.

[62] TUTORIALS, H. Dynamic malware analysis tools. https://www.hackingtutorials.org/malware-analysis-tutorials/dynamic-malware-analysis-tools/.

[63] VASUDEVAN, A., AND YERRABALLI, R. Cobra: Fine-grained malware analysis using stealth localized-executions. In *IEEE Symposium on Security and Privacy* (2006), IEEE Computer Society, pp. 264–279.

[64] VIDYARTHI, D., CHOUDHARY, S. P., RAKSHIT, S., AND KUMAR, C. R. S. Malware detection by static checking and dynamic analysis of executables. *International Journal of Information Security and Privacy 11*, 3 (2017), 29–41.

[65] WARD, M. Technology — warcraft game maker in spying row. http://news.bbc.co.uk/2/hi/technology/4385050.stm, Oct 2005.

[66] WIKIBOOKS. X86 disassembly. https://upload.wikimedia.org/wikipedia/commons/5/53/X86_Disassembly.pdf, 2013.

[67] YAN, W., ZHANG, Z., AND ANSARI, N. Revealing packed malware. *IEEE Security & Privacy 6*, 5 (2008), 65–69.

[68] YANG, W., ZHANG, Y., LI, J., SHU, J., LI, B., HU, W., AND GU, D. Appspear: Bytecode decrypting and dex reassembling for packed android malware. *Research in Attacks, Intrusions, and Defenses Lecture Notes in Computer Science* (2015), pp. 359–381.

[69] ZHANG, F., LEACH, K., STAVROU, A., AND WANG, H. Towards transparent debugging. *IEEE Trans. Dependable Sec. Comput 15*, 2 (2018), 321–335.

[70] ZHAO, Q., RABBAH, R. M., AMARASINGHE, S. P., RUDOLPH, L., AND WONG, W.-F. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In CC (2008), L. J. Hendren, Ed., vol. 4959, Springer, pp. 147–162.

[71] ZHIVICH, M., AND LEEK, T. Dynamic buffer overflow detection, Jan. 14 2012.