Cracking Solitaire

David Simonetti
University of Notre Dame
Notre Dame, USA
dsimone2@nd.edu

Thomas Mercurio
University of Notre Dame
Notre Dame, USA
tmercuri@nd.edu

Patrick Hsiao
University of Notre Dame
Notre Dame, USA
phsiao@nd.edu

Kevin Kim
University of Notre Dame
New York, USA
jkim83@nd.edu

ABSTRACT

Over 95 percent of Americans have access to the internet today and use it for various products and services. However, with the collection of sensitive personal data by service providers, there is a critical need for robust data protection measures. Cybersecurity threats, exemplified by hackers seeking unauthorized access to personal information, pose a significant threat to the economy and public trust in corporations. In 2023, the global cost of cybercrime surpassed $8 trillion.[5] This paper delves into program vulnerabilities through an exploration of notable cyber attacks in recent history as well as a successful attempt to crack Microsoft Solitaire, shedding light on how hackers exploit such weaknesses.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
David Simonetti, Thomas Mercurio, Patrick Hsiao, and Kevin Kim. 2024. Cracking Solitaire. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 NOTABLE CYBER ATTACKS

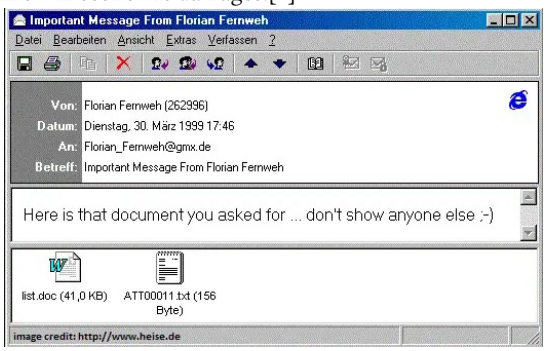
1.1 Introduction

This first section serves to provide a motivation for why software security is a topic of upmost importance. In each subsection, there will be discussion of a real world example of where software insecurity enabled a cyberattack, followed by analysis of the repercussions of the attack.

1.2 Melissa Virus (1999)

The Melissa Virus was a virus that targeted Microsoft users and exploited vulnerabilities in Microsoft Word. Created by programmer David Lee Smith, it enticed users to open their outlook email with passwords to adult websites. Once the user downloaded and opened the file in Microsoft Word, the virus sent emails through Outlook to the user's contacts with similar profane and suggestive content in the title and body. While its intent was not to steal money or information, it overloaded emails at major corporations including Microsoft, Intel, and the US Marine Corps as well as slowing down

internet access in certain areas. All in all the virus caused over $80 million in economic damages.[2]



1.3 Estonia Cyber Attack (2007)

The Estonia Cyber Attack was an attack carried out by Russian hackers on Estonian government entities and corporations. It was in retaliation for the relocation of a soviet statue to a less prominent location in the capital city of Talinn. The Russians launched a series of Denial of Service (DOS) attacks, causing major damage to government websites, banks, and companies. The Estonian government had to eventually block all international web traffic and close all digital borders. Lasting close to three weeks, recovery from the attacks cost over 8 million euros. This was the first time that a foreign nation used a cyberattack to threaten the political stability of a country, and caused many countries and the EU to review and revamp their cyber security measures.[7]

1.4 WannaCry Ransomware Attack (2017)

The WannaCry ransomware cryptoworm was a global attack that targeted using the Microsoft operating system. Led by a group of hackers called the Shadow Brokers, it leveraged stolen technology originally developed by the NSA. It encrypted information on the user's computer and made it inaccessible, demanding Bitcoin payment to recover the data or permanent deletion. In additional individual computers, it also affected the United Kingdom's hospital system as well as a Spanish mobile carrier. All in all, it affected over 200,000 devices and led to an unprecedented $7 billion in damages worldwide.[3]

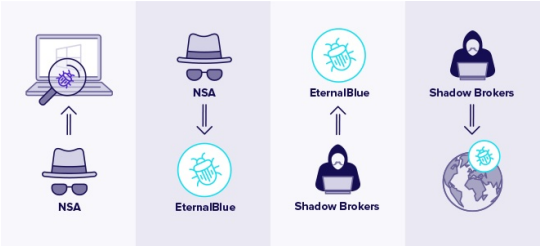
2 TOOLS USED BY HACKERS

2.1 Introduction

In this section, there will be a survey of existing tools and methods cyber threat actors utilize in order to hack into systems. Knowing how bad actors crack software is vital to providing a strong defense against such attacks.

2.2 EternalBlue

EternalBlue was a computer exploit software originally developed by the NSA to gain access to any computers within a network by exploiting vulnerabilities in the Microsoft operating system. More specifically, it targeted the 'Server message block (SMB)' protocol, which mishandled certain packets of information from malicious parties and allowed it to enter the network. It was later stolen by the Shadow Brokers who used it to carry out the WannaCry ransomware attacks. Other affected parties from what was supposed to be highly confidential NSA technology included the City of Baltimore. This brought into question the security of the NSA at protecting their technology from hackers, as well as ethical questions the agency about not disclosing a critical flaw to Microsoft for over five years.



2.3 Pegasus

The Pegasus spyware was developed by an Israeli security company that was incredibly powerful because it could be remotely installed and executed. It allowed for 'zero click exploit' on iOS devices and some Android devices. Zero click exploit meant it did not need the user to take any action for the spyware to execute, meaning they would have no idea that it was even running on your phone. Once executed, it would allow the hacker to read text messages, listen in on calls, collect passwords as well as track location. It is alleged that NSO Group, the creators of the spyware, sold it to the Saudi Arabian government which in turn used it to spy on adversaries like American journalist Jamal Khashoggi. Khashoggi was assassinated at a Saudi Arabian consulate in Turkey in 2018. Pegasus highlights the evolution of cyberattacks from innocuous email pranks and ransomware to nefarious technology that not only threatens individuals' private information and online data, but also their physical well-being, and in some cases, even lives.

2.4 Shamoon Virus

The Shamoon virus was a virus that targeted Microsoft OS users. This virus was particularly effective and harmful because it targeted the master boot record(MBR), which identifies where a computer's operating system is located. By overwriting the MBR the Shamoon Virus rendered computers virtually unusable. It was allegedly used by North Korean hackers in 2014 to erase Sony Pictures' computer infrastructure in protest of a movie called The Interview, which depicted North Korean leader Kim Jong-Un in an unflattering manner. Other affected parties by this virus included Saudi oil and gas company Aramco.

3 TOOLS TO ENHANCE SECURITY

3.1 Introduction

In this section, there will be an overview of common techniques and software that professional cybersecurity experts make use of.

3.2 McAfee

McAfee is a software designed to protect computers and digital devices against several kinds of cyber threats including viruses, online threats, ransomware, and more. It is capable of monitoring computers for signs of malicious software, and also protects devices against online threats in real time. By maintaining an updated threat database, it to stays up to date with any new viruses that may appear and affect devices. It can monitors against ransomware by identifying encryption attempts to prevent attacks like the WannaCry worm.

3.3 Firewalls

Firewalls are a mechanism to filter what gets to enter or leave your device, checking for malicious viruses or data packets that may threaten your device. There are several kinds of firewalls. Packet filtering firewalls inspects packets sent between computers to determine which should be allowed and which should be rejected. As the most basic kind of firewall, it may not be up to date with the most recent threats and viruses. Connection tracking firewalls go beyond packet filtering by monitoring active network connections. In addition to inspecting individual packets, this type of firewall keeps track of IP addresses of the associated packets to recognize where data comes from. Layer 7 filtering is the most sophisticated form of firewalls. It is capable of web filtering, can recognize FTP, HTTP, DNS, etc. and can filter based on these protocols and specific applications. This allows for more targeted restrictions and security policies.

3.4 Sandboxing

Sandboxing is the practice of executing files within a controlled environment that is isolated from the rest of the system. This environment restricts the program from accessing sensitive data that may cause damage or unintended encryption. When a potentially malicious file is detected, it is placed in the sandbox and executed under controlled conditions. If the file attempts to modify other files, access system resources, encrypt files, or establish connections with other devices, it can be removed with minimal actual damage.

4 HACKING METHODS

4.1 Reverse Engineering

An crucial process used by both hackers and security professionals to develop nearly all the tools mentioned two sections is reverse engineering. Reverse engineering is the process of being given a final form of a product with little to no information about its inner workings, and breaking it down into components to achieve a particular objective. One way to reverse engineer a program could be using a disassembler to translate machine code *back* to assembly code and analyzing it. Whether it's to exploit vulnerabilities in the iPhone to secretly monitor anyone's location, calls, and text messages or identify the origins of potentially malicious filesand prevent it from entering a device, reverse engineering plays a critical role in computer security.

4.2 Diassembly Patching

Disassembly patching is a technique used in reverse engineering to modify the behavior of software by changing its machine code instructions. Through disassembling the executable code of a program, hackers can gain insight into its inner workings. The program behavior can change in ways such as skipping certain security measures, giving access to sensitive resources, or altering files. This requires the use of disassemblers which takes in machine language and outputs assembly language. Common disassemblers include IDA and Ghidra, the latter of which was developed by the NSA and used in this project.

4.3 Buffer Overflow

Buffer overflow is a common vulnerability that occurs when a program attempts to write more input data from a user into its storage area(buffer). The danger with buffer overflow is that the program's response to overflow can be unpredictable. In the context of computer security, this can allow hackers to modify and corrupt data that should be protected. It is also capable of making an application execute specific code sent by the hackers, leading to the original user losing control of the device. While many modern operating systems have built-in protections against buffer overflow, there were several buffer overflow exploits inthe late 20th century into the early 21st.

5 SOLITAIRE

5.1 Introduction

This section provides an overview over the main topic of this paper: the video game Microsoft Solitaire. It begins with an outline of how the game is played and follows with a discussion of how the game is won.

5.2 Definitions

Because terminology of Solitaire games can vary, we provide brief explanations for key terms used in this context:

Deck: A standard 52-card deck used in playing Solitaire. It typically consists of four suits (hearts, diamonds, clubs, and spades) and thirteen ranks (Ace through King) in each suit.

Tableau: The main area of play in Solitaire where the cards are arranged into stacks. These stacks are initially dealt out in a specific layout, and the player makes moves by manipulating the cards within these stacks.

Hand: After dealing out the initial setup of the tableau in the beginning of the game, the rest of the cards will be dealt into the hand. In the hand, only the top card can be visible at a time.

Piles: The designated piles where the player builds up each suit from Ace to King. These piles serve as the ultimate goal of the game, and the player's objective is to move all cards to these piles, organized by suit and in ascending order from Ace to King.

Draw: The action of taking the top card from the hand and moving it to the discard pile. Only the top card of the hand is visible at a time.

Sequence: A consecutive series of cards of alternating colors and descending ranks. In Solitaire, sequences of cards can often be moved together as a unit, provided they adhere to the game's rules regarding card placement and movement.

Redeal: The action of resetting the cards in the hand. In some Solitaire variants, players have the option to redeal the cards in the tableau after they have gone through the deck. This allows for additional opportunities to make moves and potentially win the game. In our case, though, that is not the case.

5.3 Setup

Start by shuffling the deck and dealing out 7 stacks of cards in a horizontal row. The first stack should have 1 card, the second stack should have 2 cards (one face down, one face up), the third stack should have 3 cards (two face down, one face up), and so on, until the seventh stack which should have 7 cards (six face down, one face up).

5.4 Objective

The goal is to move all the cards to the designated piles, which are built up in suit from Ace to King.

5.5 Gameplay

You can move cards between the tableau (the 7 piles) and to the designated piles according to certain rules. Cards in the tableau can be built down by alternating colors. For example, a black 8 can be placed on a red 9. You can also move sequences of cards. For example, if there's a red 7 on a black 8, and a black 6 on that red 7, you can move all three cards together onto another tableau stack if the top card of the destination stack is the opposite color and one rank higher than the bottom card of the sequence. Empty tableau spaces can be filled with any card or sequence.

5.6 Drawing

You draw one card at a time from the deck. You can either play this card onto the tableau or the foundation, or you can keep it in your hand for later use.

5.7 Re-deal

Once you've gone through the hand, you can pick up all the cards in the hand and start back at the beginning of the hand.

5.8 Winning

The game is won when all cards are moved to the designated piles, arranged by suit from Ace to King.

6 SOLUTIONS

6.1 Introduction

This section covers how our group managed to create a cracked version of Solitaire that allows one to automatically beat the game. This section will discuss what techniques and methods were needed in order to create the cracked verison, along with how we accomplished it.

6.2 Hacking the Visibility of Cards

The first barrier towards the goal of fully automated Solitaire is that, when one plays Solitaire, most of the cards are not visible at any given time. If one wishes to implement a Solitaire solving algorithm, they must know the status of every single card on the board at any given time. Therefore, the first goal in the project should be to crack the Solitaire executable so that the initial configuration of the cards can be known.

6.3 Cracking Solitaire

The cracking of the Solitaire executable began with running the program through the NSA developed open source reverse engineering tool Ghidra [6]. Ghidra analyzes the machine code that makes up the Solitaire executable, and turns it into equivalent C code with automatically generated variable names. This is a huge boon in the reverse engineering process, as unwieldy machine code can now be viewed through the lens of much higher level C functions.

The goal of the first stage of cracking was to find the function that shuffled all of the cards in the deck whenever a new game was dealt. Once this function was found, we could hijack it in order to have it output the complete board state at the start of every new game. The shuffle function most likely called rand, so to find the shuffle function, Ghidra was searched to analyze all references to the rand function as seen in figure 1.

Location	Label	Code Unit	Context
010011f0	PTR_rand_0100...	addr MSVCRT.DLL::rand	DATA
010027d2		CALL dword ptr [->MS...	COMPUTED_CALL
01002b09	LAB_01002b09	CALL dword ptr [->MS...	COMPUTED_CALL
01006630		CALL dword ptr [->MS...	COMPUTED_CALL
01008964		CALL dword ptr [->MS...	COMPUTED_CALL
01008990	LAB_01008990	CALL dword ptr [->MS...	COMPUTED_CALL

Figure 1: All references to the rand function

Looking through all the functions that called the rand function, one in particular stood out, included as figure 2 below. One can see two nested for loops along with swapping elements in an array. This structure looks an awful lot like sorting, but it utilizes randomness. Sorting with randomness is effectively shuffling, so this is the shuffle function! To confirm this suspicion, the function was modified such that the first instruction was a RET instruction, which effectively meant the function did not do anything. The result of this change can be seen in figure 3. This is what an unshuffled game of Solitaire looks like, with cards in perfect order. From this, we were also able to determine the way in which Solitaire dealt out cards from the shuffled deck onto the board. It dealt the kings first as the bottom row of the tableau, then queens, etc, ending with the 7 of clubs as the top card on the rightmost tableau. Then we noticed that the deck was dealt in reversed order, i.e. the aces were on the bottom and the 6s on top. This will help us reconstruct the game after extracting the shuffled card order.

Now that we have found the shuffle function, we want to add our own code to the Solitaire executable so that we can output the state of the deck after shuffling. In order to do this, the approach we took is called adding a code cave. What a code cave is is a piece of code that is placed in empty space in an executable, which is then jumped to from existing program code in order to hijack the program flow.

```
for (local_8 = 0x10; local_8 < 5; local_8 = local_8 + 1) {
  for (local_c = 0; local_c < *(int *)(param_1 + 0x1c); local_c = local_c + 1) {
    iVar4 = rand();
    iVar4 = (iVar4 % *(int *)(param_1 + 0x1c)) * 0xc + param_1;
    iVar5 = local_c * 0xc + param_1;
    uVar1 = *(undefined4 *)(iVar5 + 0x24);
    uVar2 = *(undefined4 *)(iVar5 + 0x28);
    uVar3 = *(undefined4 *)(iVar5 + 0x2c);
    iVar5 = local_c * 0xc + param_1;
    *(undefined4 *)(iVar5 + 0x24) = *(undefined4 *)(iVar4 + 0x24);
    *(undefined4 *)(iVar5 + 0x28) = *(undefined4 *)(iVar4 + 0x28);
    *(undefined4 *)(iVar5 + 0x2c) = *(undefined4 *)(iVar4 + 0x2c);
    *(undefined4 *)(iVar4 + 0x24) = uVar1;
    *(undefined4 *)(iVar4 + 0x28) = uVar2;
    *(undefined4 *)(iVar4 + 0x2c) = uVar3;
  }
}
return 1;
```

Figure 2: The dissassembly for the shuffle function

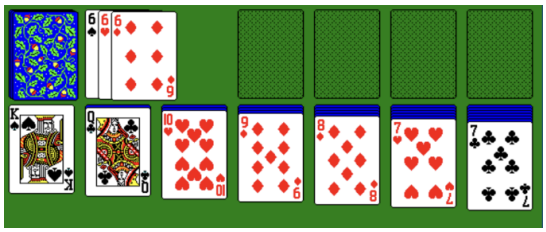


Figure 3: A game of Solitaire without shuffling

A diagram of such a code cave is included as figure 4. Essentially, we want to jump to our code cave from the shuffling function (marked A in the diagram) after the shuffling is done, output the information about all the cards to the outside work, and then return to normal program execution (marked C in the diagram). In order to make a code cave, however, we need to find some free space to place it, and that is not easily done in a compiled executable, since usually all of the free space is allocated.

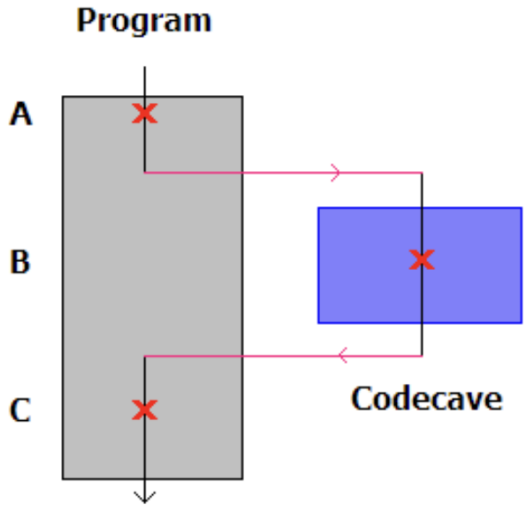


Figure 4: A diagram of a code cave

We were in luck, however, as we were able to find the perfect space for a code cave. Figure 5 shows a leftover debug function that attempts to open a debug file and write some lines into it. Replacing the first instruction of this function with all NOP instructions (can be seen in figure 6) and rerunning Solitaire shows that none of the functionality is affected, so we are safe to overtake this part of the executable for our code cave.

```
local_8c = OpenFile("sol.dbg",&local_118,0x1001);
if (local_8c != -1) {
  FUN_0100a490(local_8c,"Assertion Failure");
  FUN_0100a4d0(local_8c,param_1,param_2);
  FUN_0100a490(local_8c,"Version 3.0i");
  FUN_0100a4d0(local_8c,"Game #",DAT_0100f17c);
                    FUN_0100aaa0
00aaa0 c3               RET
```

Figure 5: Dissassembly of the debug function

```
00aaab 90               NOP
00aaac 90               NOP
00aaad 90               NOP
00aaae 90               NOP
00aaaf 90               NOP
00aab0 66 90            NOP
00aab2 90               NOP
00aab3 90               NOP
00aab4 90               NOP
00aab5 90               NOP
00aab6 90               NOP
00aab7 90               NOP
00aab8 90               NOP
00aab9 90               NOP
00aaba 90               NOP
00aabb 90               NOP
```

Figure 6: Debug function after replacing code with NOPs

Now that we have space for our code cave, we have to actually write the code to go into the code cave. Our first attempt at hacking Solitaire was as follows. In the import table for the Solitaire executable, we saw that there was OpenFile and write defined as dynamically linked functions (can be seen in figure 7). Therefore, we decided that our first attempt would be to open a file called "cheat.txt" and write out each of the positions of the 52 cards to a different line in the file. There was a problem with this approach, however, as OpenFile returns a HFILE handle while write expects a FileHandle handle. Because they use different levels of Windows file abstractions, the two functions cannot be used together.

Our second attempt preceeded as follows. Looking at Microsoft's documentation for the OpenFile function [4], we were able to figure out that it supports paths up to 128 characters long. Additionally,

Figure 7: Import table entries for OpenFile and write

the Solitaire game itself stores each card as a 2 digit number from 0-51. It does this by storing the suit of the card as its number modulus by 4 and the rank of the card as its number divided by 4, ignoring remainder. We figured that we can utilize this encoding scheme by storing all 52 cards as 2 characters each (for example, the Ace of Clubs as 00 or the King of Hearts as 51) would work out as that is only 104 characters. This does have the limitation, however, that the game must be placed in a filepath of 24 characters or less or the crack wont work. Pictured in figure 8 is the modified assembly code in our code cave that performs the magic of creating the file. We call LocalAlloc to allocate memory for a string to store the "filename", which is really the encoding of how all 52 cards were shuffled.

```
0100abb0 68 00 03 00 00    PUSH    0x300
0100abb5 68 40 00 00 00    PUSH    0x40
0100abba ff 15 a0 10 00 01 CALL    dword ptr [->KERNEL32.DLL::LocalAlloc]
0100abc7 68 01 10 00 00    PUSH    0x1001
0100abcc 89 e8             MOV     EAX,EBP
0100abce 81 e8 a0 00 00 00 SUB     EAX,0xa0
0100abd4 50                PUSH    EAX
0100abd5 81 eb 68 00 00 00 SUB     EBX,0x68
0100abdb 53                PUSH    EBX
0100abdc ff 15 84 10 00 01 CALL    dword ptr [->KERNEL32.DLL::OpenFile]
```

Figure 8: Modified assembly code to call LocalAlloc

To verify that our solution worked, we ran a test game and compared the output of the file to what showed up in the game. This game can be seen in figure 14. The file output of this game had a file name of 13002728453249511610212418260605300803220225442341483934094011382035121917310442011437471536430746332950. Effectively, this a sequence of 52 2 character numbers, so it can be read as 13, 00, 27, ... 29, 50. Using the mapping discussed earlier, we can map from these numbers to actual cards. We also discussed the procedure with which the game uses to deal the cards from the deck onto the board. Combining this, we can verify that the last card with value 50 (King of Hearts) should be on top of the leftmost pile on the tableau. Also, 15 (4 of Spades) should be placed on top of the second from leftmost tableau pile. Finally, 13 (4 of diamonds) should be on the bottom of the deck. All of these can be visually verified in figure 9.

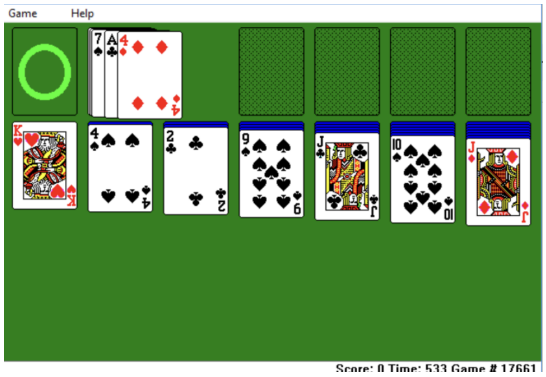


Figure 9: Test game to verify the efficacy of our solution

6.4 Preliminary Algorithm/Game Architecture

Before the implementation of the algorithm, we chose to create a "hacked" simulation of the game so that we can carry out the algorithm that we developed.

6.4.1 Board Setup. In order to setup the simulation, we first have to setup the "hacked" tableau. In order to figure out the cards that will be laid out, we used the method explained earlier to lay out the tableau. In order to check that the title text input is valid, we implemented a method designed to parse a given file name string and extract pairs of two-digit numbers, which represent card codes, from it. The method initializes an empty list called `card\_codes` to store the extracted numbers. It then iterates through the characters of the file name string from right to left, starting from the second last character. At each iteration step, it checks if the current character and the previous character form a valid two-digit number by using string slicing to extract a substring containing two characters and checking if it consists entirely of digits using the `isdigit()` method. If a valid two-digit number is found, it is appended to the card_codes list. After each iteration, the index is decremented by either 2 (if a valid number is found) or 1 (if not) to move to the previous pair of characters for the next iteration. Once the iteration through the file name string is complete, the method returns the list of extracted card codes. To create an understandable representation for the user, we setup number to suit and suit to number mappings so that we can easily represent the cards. This method is utilized within the `\_\_init\_\_ constructor` of the Solitaire class to extract the initial setup of card codes from the file name, facilitating the initialization of the game's tableau and other components. We then distribute these cards into the tableau. The tableau is represented as a list of lists within the Solitaire class. Each inner list corresponds to a tableau pile, with the outer list containing all seven tableau piles. Each pile in the tableau is a stack of cards, where the top card of each pile is the visible card. The remaining cards will be placed in the hand. These cards are all visible to us "hackers".

6.4.2 Hitting/Re-dealing the Hand. The movement of what the visible card on top is actually represented by a pointer. After each hit, the pointer moves through the hand. In this way, we can see where the top is while being able to access all the cards in the hand

as a hacker. Once the cards are all depleted, we can just reset the pointer to the end of the list to re-deal the hand.

6.4.3 Overview of Movement and Validity. Because of the many different ways that a player can move a card, we decided to consolidate the movement locations to numbers so that we can easily understand where the card is going and where it's coming from. We set it up so that 0 was hand, 1-7 were the columns of the tableau, 8 was the clubs pile, 9 was the diamonds pile, 10 was the hearts pile and 11 was the spades pile. We then used the `move` method to actually move the cards. It serves the purpose of executing moves of cards from one location to another within the game's tableau and piles. The method begins by checking the validity of the proposed move using the `check\_validity` method. If the move is deemed invalid, the method simply returns without executing any further actions. Upon confirming the validity of the move, the method proceeds to carry out the card movement based on the specified source and target locations. Like the hand, the visible cards of each tableau stack is represented by a pointer so that we can determine whether we can move the card(s) or not.

6.4.4 Move function. If the source location is the player's hand, the method removes the top card from the hand and places it either onto a tableau column or into one of the four piles, depending on the target location. In the case of moving cards from the tableau to the piles, the method transfers the visible cards from the selected pile to the target tableau column. Conversely, if the target location is one of the four piles, the method takes the top card from the specified tableau column and adds it to the designated pile. For moves within the tableau columns, the method allows for the transfer of a stack of cards from one tableau column to another, adjusting the visible card pointers accordingly. After executing the move, the method ensures that the visible card pointers for both the source and target tableau columns are appropriately updated to reflect any changes in the game state. Overall, the move method plays a crucial role in managing the movement of cards within the Solitaire game, contributing to the game's progression and ensuring adherence to its rules.

6.4.5 Check_Validity function. The `check\_validity` function encompasses several cases to ensure that moves adhere to the game's rules. Firstly, it checks if the source and target piles are the same, which would constitute an invalid move. Secondly, it verifies whether the source or target pile is out of bounds, which would also render the move invalid. Moving on, if the target pile is one of the four designated piles (`clubs, diamonds, hearts, spades`), the function evaluates the move's validity based on the cards' suits and ranks. It ensures that cards moved from the tableau or the player's hand match the suit of the target pile and are appropriately ranked relative to the top card in the target pile. Additionally, it permits pile-to-pile moves as they are inherently valid. Lastly, when moving to the tableau, the function confirms that cards moved from one tableau pile to another or from the player's hand to the tableau adhere to the game's rules regarding alternating card colors and consecutive card ranks. It also ensures that kings are only placed on empty tableau piles. In each case, if the move is deemed invalid, an error message is printed

6.4.6 Play function. Now that we have all the functions to play the game, we can actually represent the cards in the play function. To make our task easier, we designed a visual representation of the board through print statements and clear statements to explain what each list represented. In this way, we can see the game every step of the way. Now, we can move on to designing an algorithm that will find a solution for our game!

6.5 Algorithm Research

To come up with the algorithm to crack solitaire, we began by playing the game ourselves, thinking of ways to turn our human intuition about the game into machine code. During this time, we realized that cracking the game was hard, and there didn't seem to be an algorithmic way to approach the problem other than brute-force. Surveying related work, we came across a research paper called "The Winnability of Klondike Solitaire and Many Other Patience Games," written by Charlie Blake and Ian P. Gent at the University of St. Andrew's in the United Kingdom. From this paper, we learned that the Draw 3 version of Klondike solitaire is winnable 81.9% of the time, while the Draw 1 version, which we used, is winnable 90.5% of the time. In their paper, Blake and Gent outlined their algorithm for solving the game, which involved a "depth-first backtracking search solver over the state space of legal card configurations" [1]. Taking inspiration from that, we began to develop our algorithm.

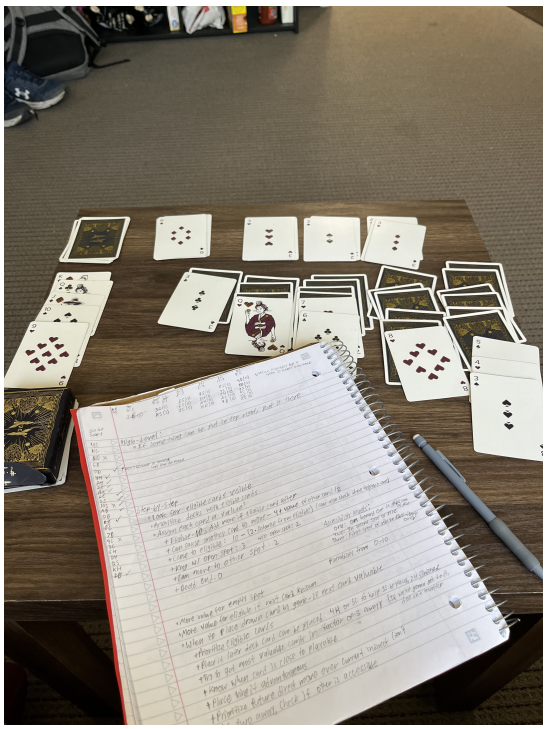


Figure 10: Brainstorming algorithm implementation

6.6 Algorithm Implementation

After researching, we decided to go with a backtracking algorithm that uses depth-first search (DFS). To do so, we came up with four different moves that can be done out of each board state, in decreasing level of priority:

(1) Move card from hand or tableau to pile
(2) Move card(s) from tableau to tableau
(3) Move card from hand to tableau
(4) Draw from hand

The prioritization of these moves corresponds with how most players would approach a game. Moving a card to the piles gets you closer to winning the game than does moving a card or cards from tableau to tableau, which, in turn, is more valuable than moving from the hand to tableau (in general).

In calling the algorithm, first, we create a game board based on the most recent file in the directory. From there, we call our function `simulate_game()`, which takes in the game board, a `verbose` option for debugging, and an output file. Going into `simulate_game()`, we first have set global variables like `dead_ends`, `total_moves_tried`, `used_boards`, and `boards_repeated` to 0. After a few print statements and a check if we've hit too many dead ends to need to continue, we go into the algorithm itself.

First, we check if any card can be moved from the hand or the tableau to a pile, using the `board.check_validity()` to see if it can be done. If so, we move the card and create a `board_string` to check if we have tried that board configuration before. If not, we complete the move, add it to the current list of steps, checking if the game has been won. If it hasn't been won, we recursively call `simulate_game` with the new board configuration, steps, and consecutive draws. The algorithm continues from there, recursing the depth of the tree. If there is no card that can be placed in one of the piles, we check if anything can be moved from tableau to tableau, executing a similar process if so. One note here is that we only check for moving the top cards and all those beneath it from tableau to tableau, because in most cases, moving the bottom card without being able to flip a new card is a pointless move. If we did add that move, we predict that it might increase our win rate by a somewhat negligible margin, so we decided that was unnecessary to include. Getting through that step with no possible move brings us to moving from the draw card to the tableau. If possible, we add it to the steps and recurse from there. The final possibility is drawing from the hand, which should only be done if we have not drawn the whole deck in a row. To check this, we use the `consecutive_draws` parameter, which is reset each time we perform one of the other moves to change the game board. It is incremented by one every time we draw. If we have drawn enough times to get through the whole hand, we have reached a dead end, and that round of `simulate_game()` returns false. We increment the counter `dead_ends` to keep track of how many of them we've run into. At this point, the call stack will jump back to where the current `simulate_game()` was called from, backtracking and trying the next possible move.

If a game is won, the moves are outputted in a file with each line representing a move in the form of three integers. The three numbers represent `(source, target, card_moved)`: the `source` is a number between 0 and 7, inclusive, representing the hand or one of the tableau stacks where the card is coming from; the `target` is a number between 1 and 11, inclusive, representing the destination as one of the tableau stacks or finished piles; and the `card_moved` comes into play in the tableau, representing which card from the stack is being moved. Drawing from the hand is represented as `(-1, -1, -1)`. In turn, this output files is then read in by the program used in the GUI, turning our solved algorithm into mouse clicks that win the game of solitaire.

Figure 11: Example of the first few lines of an output file–this run took 125 moves

7 AUTOMATING SOLITAIRE

7.1 Introduction

This section covers how our group created the automation and visual representation of the automation that we implemented. This section expands upon how we used the output of our moves to actually "hack" the Solitaire game.

7.2 Windows API and Automation

The Windows API was used heavily to automate interactions with the Solitaire game window. Initially, it locates the Solitaire window using the `FindWindow` function by searching for a window with the specified title. Once found, it brings the Solitaire window to the foreground and sets it as the active and focused window using functions like `SetForegroundWindow`, `SetActiveWindow`, and `SetFocus`. For simulating mouse input, the code utilizes the `SendInput` function to send input events to the system's input queue. This is employed to simulate left mouse button clicks using the `left\_click` function and mouse drag actions via the start_drag and end_drag functions.

These functions construct `INPUT structs` with appropriate flags to specify the type of mouse input event being simulated. One of

```c
void left_click() {
    // set up INPUT struct to indicate a left mouse down click
    INPUT clickInput = {0};
    clickInput.type = INPUT_MOUSE;
    clickInput.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;

    // Send the input
    SendInput(1, &clickInput, sizeof(INPUT));

    Sleep(movement_time);

    // clear INPUT struct
    ZeroMemory(&clickInput, sizeof(INPUT));
    clickInput.type = INPUT_MOUSE;
    clickInput.mi.dwFlags = MOUSEEVENTF_LEFTUP;
    // Send the input
    SendInput(1, &clickInput, sizeof(INPUT));
}
```

Figure 12: Code utilizing Windows.h API to simulate a leftclick

```c
void center_mouse_on_deck() {
    RECT rect = {0};
    GetWindowRect(solitaire_handle, &rect);

    Sleep(movement_time);

    SetCursorPos(rect.left + 50, rect.top + 100);
}

void center_mouse_on_deck_cards() {
    RECT rect = {0};
    GetWindowRect(solitaire_handle, &rect);

    Sleep(movement_time);

    SetCursorPos(rect.left + 160, rect.top + 100);
}
```

Figure 13: Code utilizing Windows.h API to move the mouse

the biggest uses for Windows API was for positioning in the actual game. To position the mouse cursor accurately on specific elements within the Solitaire window, functions such as `SetCursorPos` are used. These functions calculate the cursor's position based on the window's coordinates obtained using the `GetWindowRect` function.

7.3 GUI

For the GUI, we utilized `Tkinter`, a GUI toolkit, to create a graphical interface for automatically solving and dealing new games of Solitaire. It imports necessary modules such as `tkinter`, `subprocess`, and `ttk`. The `solve\_game()` function runs an external Python script called `alg.py` to solve the current Solitaire game. If no solution is found, it displays an informational message; otherwise, it executes another external executable named `autosolver.exe`. The `deal\_new()` function deals a new game of Solitaire by running `autosolver.exe` with the "deal" argument. The script launches the Solitaire game window using `subprocess.Popen()`, creates the main `Tkinter` window with a title, configures button styles, creates buttons for solving and dealing new games, and packs them into the window. Finally, it enters the `Tkinter event loop` to keep the GUI responsive.

Figure 14: The GUI that allows users to solve Solitaire automatically

8 DISCUSSION

8.1 Challenges

In our task, we encountered several challenges. Firstly, working within the constraints of limited lines in Assembly to create a code cave presented a significant hurdle. This limitation necessitated efficient coding within the available space. Additionally, implementing a Backtracking Algorithm proved to be challenging due to its intricate logic and complexity. Creating and running such an algorithm required meticulous attention to detail and thorough testing to ensure correctness. Moreover, managing time and processing power posed difficulties, particularly when executing Python scripts that demanded substantial computational resources. This strain was evident in instances where the CPU usage peaked at 99%, potentially leading to performance issues and slowdowns. Lastly, debugging proved to be a tough task. During our debugging, we had to check many edge cases and look through a ton of lengthy lines of print statements and numerous errors.

8.2 Distribution of Work

David worked on the cracking of the Solitaire game as well as the game automation. Patrick worked on implementing Solitaire in Python as well as reading in the output files of the crack so that our algorithm could ingest the card ordering. Thomas and Kevin both worked on implementing the Solitaire solving algorithm.

9 CONCLUSION

9.1 What We Learned

Through this project, we learned a lot about the processes of hacking and the emphasis on computer security. We learned about different events throughout history that required the need for computer security and the techniques and strategies that were needed for this use case. Computer security is a very relevant field and will only increase in relevance. After "hacking" Solitaire, we not only found out how computer security needs to be implemented for all applications, but also explored the process of protecting sensitive information and guarding against potential cyber threats. Our experience delving into a seemingly simple game like Solitaire highlighted the importance of robust security measures in every aspect of software development and usage. Furthermore, it emphasized the necessity for continuous learning and adaptation in the face of evolving hacking techniques and technologies. As we move forward, we are committed to applying the knowledge gained from this project to contribute to the ongoing advancement of computer security, ensuring the integrity and safety of digital systems for the world.

9.2 Future Work

In future work, there are several avenues to explore. One potential direction is to apply the algorithm developed in this project to more complicated games beyond solitaire. We learned a lot in hacking the game and using the method we took can possibly be applied on other gaming scenarios. Additionally, optimizing CPU usage represents a crucial area for improvement, particularly in resource-intensive tasks like running Python scripts. Implementing strategies to streamline code execution, reduce computational overhead, and optimize algorithms could help alleviate processing bottlenecks and enhance overall system efficiency. Another area of focus could be minimizing the difference in success rates between using one draw and three draw options in solitaire games. Investigating and refining the algorithm to achieve more balanced success rates across different draw settings could improve user experience and gameplay consistency. Our current success rate is a little above 50%, so if we can improve that, we can get closer to the actual game playability rate. By addressing these areas in future research and development efforts, we can advance the capabilities and performance of our system, enabling it to tackle more complex challenges and deliver more robust solutions.

REFERENCES

[1] Ian P. Gent Charlie Blake. 2023. *The Winnability of Klondike Solitaire and Many Other Patience Games.* Retrieved May 7, 2024 from https://arxiv.org/pdf/1906.12314
[2] FBI. 2024. *The Melissa Virus.* Retrieved May 6, 2024 from https://www.fbi.gov/news/stories/melissa-virus-20th-anniversary-032519
[3] Kaspersky. 2024. *What is WannaCry ransomware?* Retrieved May 6, 2024 from https://usa.kaspersky.com/resource-center/threats/ransomware-wannacry
[4] Microsoft. 2024. *OpenFile documentation.* https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-openfile
[5] Steve Morgan. 2024. *Cybercrime To Cost The World 8 Trillion Annually In 2023.* Retrieved May 6, 2024 from https://cybersecurityventures.com/cybercrime-to-cost-the-world-8-trillion-annually-in-2023/
[6] NSA. 2024. *Ghidra.* https://ghidra-sre.org/
[7] Council on Foreign Relations. 2024. *Estonian denial of service incident.* Retrieved May 6, 2024 from https://www.cfr.org/cyber-operations/estonian-denial-service-incident