Semester Project Report

Alex Issing & Fiona Mustard

May 1, 2024

1 Introduction

1.1 Pokemon Silver

We chose to reverse engineer the game Pokemon Silver for our semester project. We had both played the series before and knew that there was the potential to modify key values. As such, it seemed like a great game to test our reverse engineering skills on.

The plot of the game is as follows: a teenager is given a Pokemon (a portmanteau for Pocket Monster: a creature with special powers that can be captured, stored and summoned by its owner) to help out the local scientist, Professor Elm. As this teenager, the player travels around the country, exploring and completing tasks. While exploring, the player populates their Pokedex, also provided by Professor Elm, which keeps track of the Pokemon a player encounters.

The objectives of the game include 1) fighting other trainers' Pokemon, 2) collecting badges for fighting specific Pokemon battles, 3) defeating the best trainers (and their Pokemon) in the region to become the champion, and 4) capturing more Pokemon. (Hence the Pokemon slogan, "Gotta catch 'em all!")

Capturing 'all' of the Pokemon, however, is hindered by a disparity between the number of available Pokemon in the game and the number of Pokemon listed in the Pokedex. Pokemon Gold, the sister game to Pokemon Silver, has some unique Pokemon not found in Silver, it just as Silver has some Pokemon that cannot be found in Gold. The third game from the series, Pokemon Crystal, has additional features and items along with its own exclusive set of Pokemon. This makes it seem impossible to complete the Pokedex without trading between games. However, while Crystal and Gold were not tested with our script, at minimum Pokemon Gold's structure (which is similar to Silver's) should let the developed script manipulate access to the Pokemon in that Pokedex as well. As such, there is a way to get every Pokemon in a single game of Silver, and possibly of Gold.

1.2 Project Goals

As mentioned before, a major part of the game is to battle our own Pokemon against other Pokemon (belonging to other trainers or encountered in the wild). Each Pokemon has visible and hidden stats that determine how much damage its move will do, who moves first, etc.

With this in mind, we set out to change the selection of initial Pokemon offered by Professor Elm along with their stats, in order to make the strongest Pokemon in the game. We defined the strongest Pokemon as having stats so high that it could easily defeat opposing Pokemon in as few moves as possible while sustaining minimal damage.

To test the mettle of our "unbeatable" Pokemon, we let each attempt battle with Red's team of Pokemon. Red is a trainer from the original Pokemon game who sits at the peak of Mt. Silver and one the strongest, if not the strongest opponents in the game. He has a well-rounded, high-level team of Pokemon that serve as good foils for our purportedly unbeatable contender. Thus, if our Pokemon can beat Red's entire team of Pokemon, it could beat the teams from other trainers, and any solo wild Pokemon.

We further extended the goal to include inventory editing, which would let us maintain a undepletable mount of healing resources so we could consistently heal our Pokemon from whatever minor wounds it incurred during "Pokebattles." In order to quickly modify the values in theses goals, we plan to develop a script that can be applied to save states. This will make sure that users can effectively modify the values without having an in-depth knowledge on the RAM and ROM addresses of the game.

2 Process

2.1 Running the Game

We found a version of the extracted ROM for Pokemon Silver on the Internet Archive in a set of Game Boy Color ROMs. We searched for emulators that could run the game and decompile it while running. Initially we found a repository called gem by bassicali which played the game while monitoring all values of the virtual Game Boy in a nice UI. Unfortunately, we could not build gem on either of our machines. (This was likely due to it being developed in c++ with solution files that were not included in the initial installation repository.)

While trouble-shooting gem, we found its predecessor, a program called BGB Emulator Debugger, which ran without issue on both of our operating systems. As visible in Figure 1, it shows the decompilation of the game, active memory, flags, and the game screen; so it provided both the functionality and insights we needed to analyze and manipulate the game.

Although BGB was good for playing Pokemon Silver and performing dynamic analysis, we still needed a static analysis tool; we opted to use the decompiler Binary Ninja (Binja) by Vector35. We chose Binja because we were used to its environment and because it has a plugin for reading Game Boy

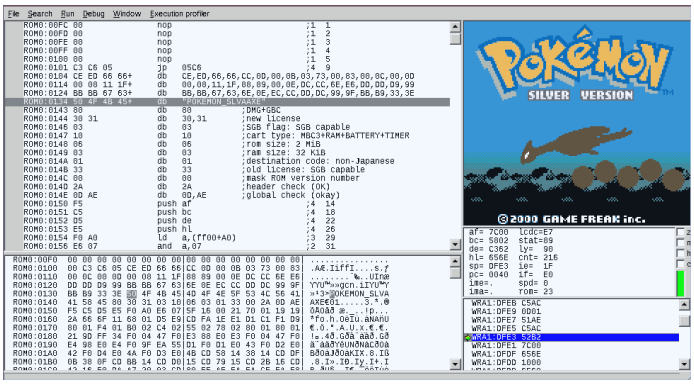


Figure 1: BGB Debugger Running Pokemon Silver

ROMs. Viewing the decompiler's reconstructed functions was easier than reconstructing them manually from the assembly code in BGB's window.

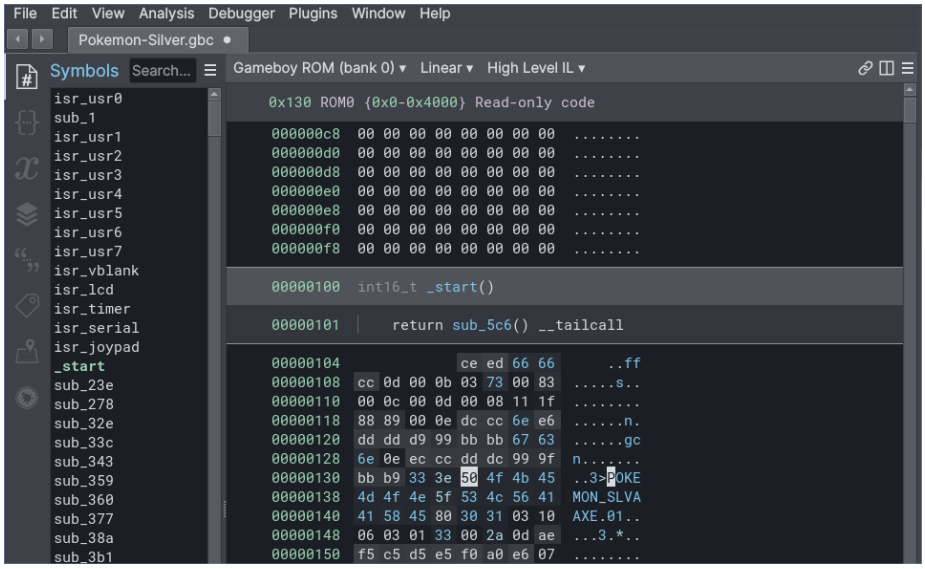


Figure 2: Binary Ninja viewing the Pokemon Silver ROM

2.2 Pokemon Stats

Since our primary objectives toward creating an unbeatable Pokemon involved 1) choosing a specific Pokemon as our starter and 2) being able to modify the Pokemon's stats so that it would be the strongest Pokemon, our first strategy was to monitor the WRAM, writable memory, for the initial Pokemon received

from Professor Elm. This was a much larger task than we expected since WRAM takes from **0xC000** to **0xDFFF**.

We used the VRAM viewer to find out how to change the position of the character for future use. In doing so, we noticed how the letters on screen were part of a series starting at **0x80** - similar to ascii starting at **0x65** for "A". Using this, we saved the state of the game ("freezing" the RAM to be viewed in Binja) and searched for hex strings. For nostalgic reasons, Alex's contract for this project required that the player character be named ACE(**0x808284**) and Cyndaquil be named FIRESHELL(**0x858891849287848B8B**).

Searching for FIRESHELL led to **0x210A**, as seen in Figure 3. In BGB, this places the address at **0xDB8D**. [1]

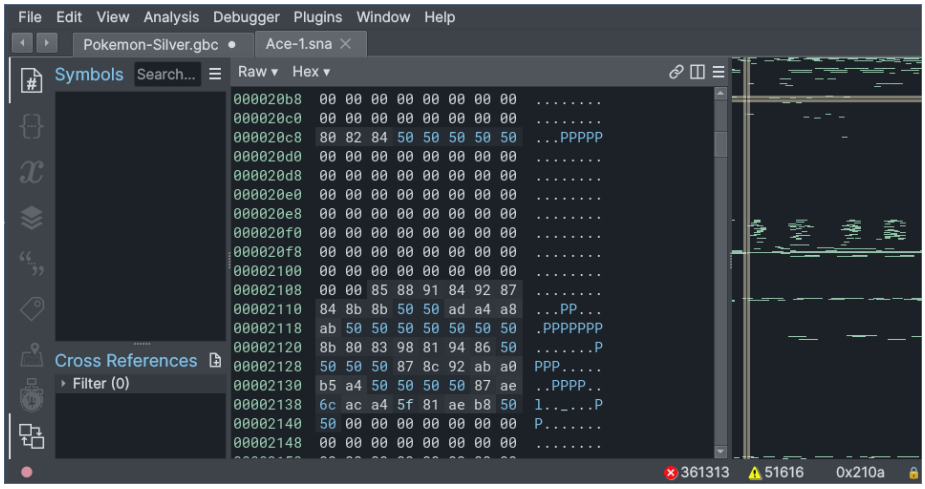


Figure 3: Trainer names starting at **0x20C8** and Pokemon names starting at **0x210A**

Looking around that address, there were many zeros. This meant that the nicknames were stored in a different location from the Pokemon's stats. Luckily, they were not stored far from each other (more like directly above), as the following struct shows:[2]

```
struct Party{
    Total in Party [1 byte]
    Pokemon in Party [6 bytes] (1 byte per Pokemon)
    End of Party List [1 byte]
    for x in range(1,6):
        Pokemon x Stats struct [48 bytes]
```

[1] This is the equation to go from BGB to the save state address, including the transition: `(offset - 0xC000) + (hexFind(filename, "WRAM")+0x9)`

[2] For more information on the actual addresses and other notes, look at the README.md on Github

```
    for x in range(1,6):
        Pokemon x trainer name [8 bytes]
    for x in range(1,6):
        Pokemon x nickname [11 bytes]
}
```

Once we found the structure of the party system (a "party" here refers to the lineup of Pokemon that our player character can summon to battle another Pokemon), we had to determine which addresses stored what stat in order to modify the correct variables.

The initial strategy was to modify a value and see if it changed anything in the stats readout when inspecting the Pokemon. This scattershot approach let us identify the visible stats, but not the hidden ones. Ultimately, we identified the hidden stats when we discovered that certain nearby values were adjusted automatically after combat, as Bulbapedia later confirmed. This let us create the following struct for each of the Pokemon's stats:

```
struct Stats{
    Pokemon [1 byte]
    Held Item [1 byte]
    Moves [4 bytes] (1 byte per move)
    ID Number [2 bytes]
    Experience [3 bytes]
    HP EV [2 bytes]
    Attack EV [2 bytes]
    Defense EV [2 bytes]
    Speed EV [2 bytes]
    Special EV [2 bytes]
    Attack/Defense IV [1 byte]
    Speed/Special IV [1 byte]
    Move PPs [4 bytes] (1 byte per move)
    Mood [1 byte]
    Pokerus [1 byte]
    Caught Data [2 bytes]
    Level [1 byte]
    Status [1 byte]
    Unused [1 byte]
    HP [2 bytes]
    Max HP [2 bytes]
    Attack [2 bytes]
    Defense [2 bytes]
    Speed [2 bytes]
    Special Attack [2 bytes]
    Special Defense [2 bytes]
}
```

Figure 4 shows the Party struct with a level 5 Cyndaquil(**0x9B**) that was

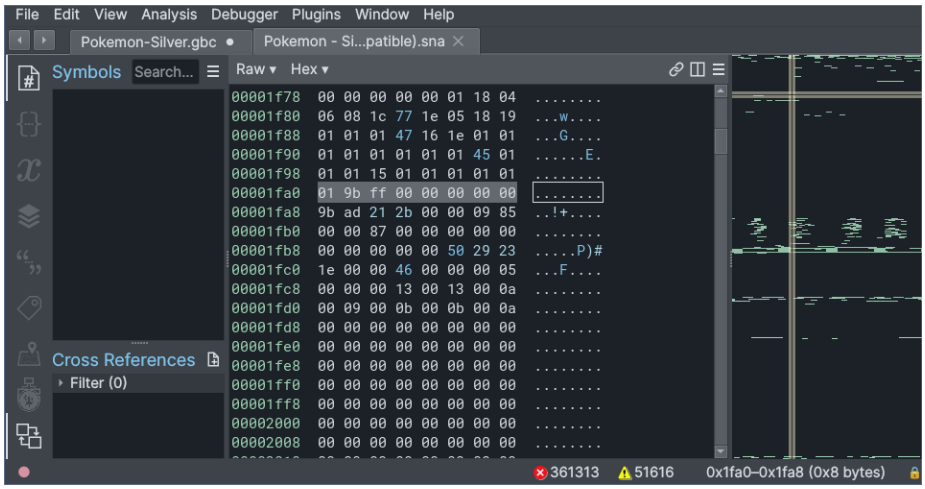


Figure 4: Start of the Party struct highlighted, with Cyndaquil's Stats struct starting at **0x1FA8**

given to us by Professor Elm. The eight bytes of the party list are highlighted. Immediately following the party list begins Cyndaquil's Stats struct.

With these structs mapped, we had all the addresses needed for the script.

As you can see, the very first element in the Stats struct stores a value that represents the Pokemon in question; the same value would also stored as one of the Pokemon in the Party struct detailed above that. While finalizing this paper, we learned that the value stored in the aforementioned Party struct is the one that determines the weaknesses, move set, experience total, etc. that are hard coded in the ROM data. The value stored in the Pokemon's stats struct appears to only change the sprite used during battle.

2.3 Inventory Management

Finding the inventory addresses was less difficult since the party was already mapped and took up a sizeable chunk of the available space. Given the structure of the party's array, we assumed the game's inventory would be in a similar format like:

```
struct Inventory{
    Total Items in Bag [1 byte]
    for x in range(1, Bag Size):
        Item [1 byte]
        Amount [1 byte]
    End of Bag [1 byte]
}
```

To confirm this expectation, we tracked changes in inventory before and after

the scene in the game where Professor elm's assistant gives the player a healing potion. Upon parsing the save state in Binja for the string **0x01**01FF**, we confirmed that the Inventory struct we posited was indeed accurate.

Figure 5 shows the items bag (highlighted) right before it indicates the number of items in the bag; the array is terminated with **0xFF**.

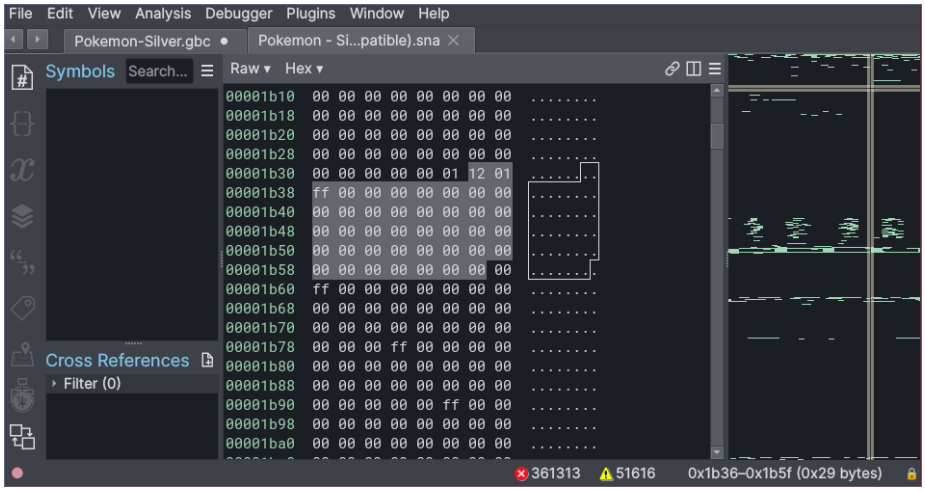


Figure 5: Highlighted items bag with 20 slots

Stored directly after the first game's item bag are a few other item bags in which more inventory can be stored, but which just happens to be empty at the start of the game. Based on expected items in that bag (determined through game play), these bag can hold keys, Pokeballs, or stored items. It stands to reason that the types of items these bag can hold are sorted on pick up while their amounts remain variable. (In the initial production of the script, we only implemented the main items bag to make sure the addresses were right and the values were changing. Eventually the previously mentioned bags were implemented in a cycling manner so as not to disrupt the script's UI too much)

Fans of the series will know that there is an additional bag that we did not mention, which would be the TM/HM bag. This bag holds special items that can be used to teach your Pokemon a specific attack or move (if the Pokemon is programmed to learn it). The TMs (Training Machines) break after a single use; the HMs(Hidden Machines) do not break and the moves can be used when exploring the world. The HMs require specific gym badges to be accessed. Unlike items in the other two bags,[3] the items in this bag are not stored in an array. The TM/HM bag has the items already hard coded into positions. Changing the byte to a non-zero value will make it appear in the bag with the specified quantity, or none specified for HMs. We only noticed the memory

[3]The keys bag will always assume the amount is **0xFF**, as it lacks the byte to store an amount. Importantly, it is still stored in an array like the main bags

location of this bag, above the items bag, after receiving a TM from one of the gym leaders.

With that, all of the bags and PC storage were located and therefore malleable, so long as they could be indexed by the script and the modified save states could be loaded.

2.4 Miscellaneous Values

This section briefly discusses miscellaneous addresses found during testing.[4] Although we already listed our main objective at the beginning, we wanted to develop the script beyond the semester goals so it could modify any given value. First on the list were gym badges.

Each of the game's two regions has 8 leaders corresponding to a bit flag in the byte of space allocated for gym badges. This means that the first leader adds **0x01** to the address, and the sixth leader adds **0x20**. Assuming the player somehow only has the two badges, then the address stores **0b00100001**.

Locating the badges is important since the player needs specific badges in order to use the HMs and progress the story. We identified the addresses after beating a gym leader and receiving a badge in one of the memory locations we were tracking.

The Pokedollars and casino coins are stored directly in front of the gym badges. We assumed that important player data would also be stored in this area because the TM/HM bag address directly followed the Kanto badges mentioned previously. Storing the Pokedollars used 3 bytes for the player's wallet and another 3 bytes for the amount given to the player's Mom.

The value for repel determines the player's ability to repel wild Pokemon, who would otherwise challenge the player to fight them over and over. The repel value stores a single byte with values decreasing by 1 with each step you take; so, at maximum, it takes 255 steps until it runs out.

The repel memory address is located next to the 4 bytes that control the cities the player can fly to. Actually mapping the fly values proved difficult, as seen in Figure 6. We essentially kept changing the values in the decompiler and checking which cities the player could fly to. The fly value appears to use a similar bit flag idea as the gym badges. As with our previous attempts, after trying different values, we eventually settled on **0xFFFFFFFF** since that allowed us to fly anywhere in the region.

The last miscellaneous value manipulated in the script corresponded to the relative location of the player in the current map, which changed every time the player moved. Manipulating the value was not very beneficial for traveling across the map, since that required foreknowledge of exact coordinates of destinations, and redrawing the memory section that hold the barriers. Instead, we tracked down the memory addresses responsible for holding the data that allowed players to fly between cities. (We discovered later that this local map movement feature could be used to place the player on top of NPCs, allowing you to skip battles or

[4] More information about each is located in the README.md of the github repository

jump over the engineer that prevents you from using the train before repairing it.)

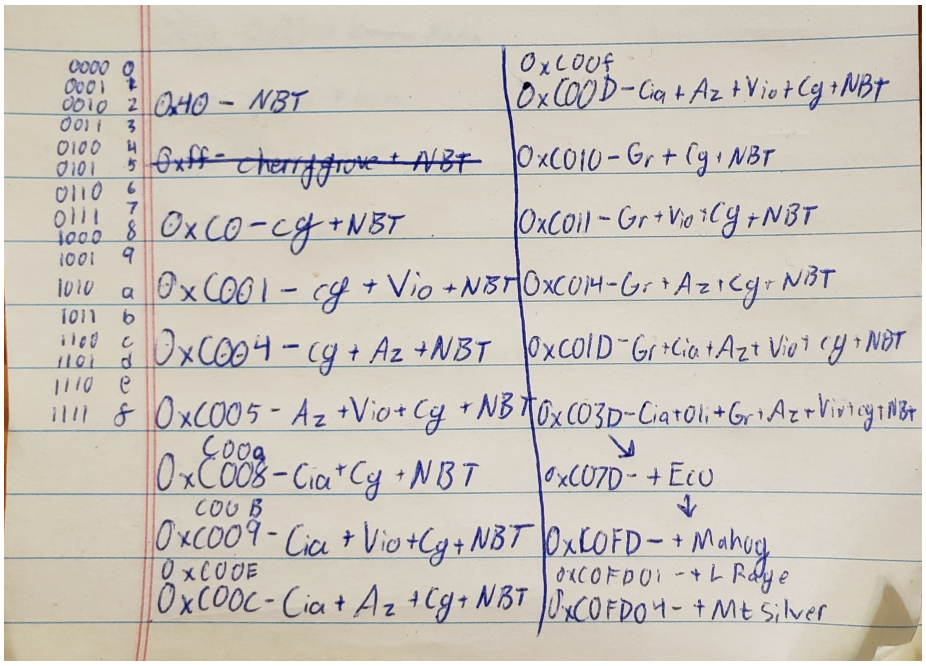


Figure 6: Notes on which values fly to which cities in Johto

2.5 Python Script

2.5.1 Development

As stated in our project proposal and demonstrated during our presentation, our goal was to create a script that would make changing the values of the game easy; we didn't want to have to mess with the WRAM every time we wanted to manipulate a Pokemon's stats. The final script allows just that, and is available on Github.

Initially, we wrote the framework for the script to be able to accept two types of files: the Game Boy's save file **.sav**(save), or BGB's save state **.sna**(state). As such, we developed a way to check the headers of the file, like the magic numbers for ELF. BGB's states always began with BGB, but the save files did not have any specific identifiers, so we resorted to checking the file extension.

Even though the framework still exists, managing saves has not been implemented. Since we were already using states to get the RAM addresses, a lot of the addresses were mapped for it. The saves on the other hand were not as one-to-one as the states were. (This is expanded on in the Future Works section(3).)

As there could be two sets of every address and additions, we wrote all of the addresses into a separate JSON file that could be imported to the main Python program. This program also contains all of the conversions between Pokemon, moves, items, and their integer values. All of the values associated with the Pokemon and the player's party were identified manually by writing the corresponding hex value into the inventory or party slot. The format of the file also makes it easy to add more addresses (like those mentioned in the miscellaneous section).

Early on while tracking down the values in the states, we used the value **0xBA82** to convert the RAM value we put in the JSON to the state, such that `offset-0xBA82` was the conversion function. We ran into trouble with that equation when one of us renamed the ROM file, subsequently breaking the script; BGB saves the ROM name and the number of characters, which changed when the file was renamed. We tried recalculating the value needed to offset the addresses using the number of characters; however, the states still would not work. Ultimately, recalling that the state is a frozen version of the memory that needs a special indicator of where the WRAM starts, we used this as an anchor. When running in BGB, the WRAM starts at **0xC000**, so we incorporated its start "value" into: `(offset - 0xC000) + (hexFind(filename, "WRAM")+0x9)`. This update allowed us, and future users, to have the ROM named anything and still be able to modify the correct values.

For the design of the script, we did not want to limit the Pokemon that the user could add to their party; therefore, the only limit imposed on the user is the max number of bytes in which their selected value (representing the Pokemon) can be stored. While testing the stats, we did encounter buffer overflow issues; and the miscellaneous values in particular tended to reject user input in favor of maxing out the bytes. While testing the Stats struct for moves, we found **0x39** to be the highest we could enter without overflowing. (This is evident in Figure 9 with the PP for all the moves are set to **0x39** when changing them). As an example of the script's modification ability, we created a custom Ho-Oh in Figure 7, which shows HostessTre with very high stats, except for special defense. This made them very susceptible to attacks from a wild, much lower-powered Rattata that should have been an easy victory.

In an effort to not limit the user's ability to glitch the game, this is the function used when checking the inputs: `int.to_bytes(max(min(int(inp), 2**(8*byteSize)-1), 0),byteSize, byteorder='big')`[5]. Using this, we only limit integers to the byte bounds. There are also statements that all the inputs to be in hex or binary, as some values are better in those formats, like when aiming for a shiny Pokemon. (We found the memory location that determined "shininess" because there is a shiny Pokemon guaranteed to appear in the game.)[6]

We mentioned earlier that initially only the manipulation of the items bag was implemented. This was a simple bag to manipulate due to the finite number

[5]The Game Boy is a Big-endian system that runs on an architecture similar to Intel 8080.

[6]README.md has notes on how to create a shiny Pokemon. Also it has recommended values for the stats; but again, it is not enforced.

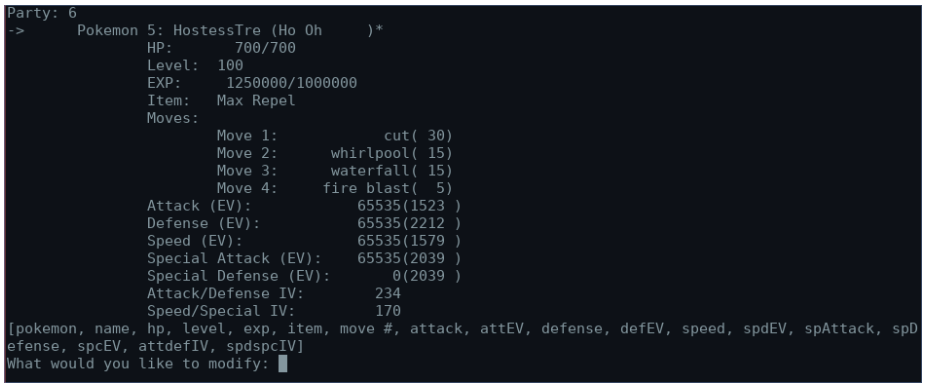


Figure 7: Party Pokemon editing UI: HostessTre

of slots (20) and the corresponding amount value associated with each item. The key bag was specially implemented to use null in the amount slot so that it could be usable in the same fashion as the items bag. The TM/HM bag used a similar strategy, but on the item value (recall that all of the TM/HM are already hard coded to a corresponding TM or HM). The bags' visualizations were also adjusted to account for a non-editable value. Since the TM/HM is not stored in an array, a random value is grabbed for the indicator of how large the bag is, as seen in Figure 8.

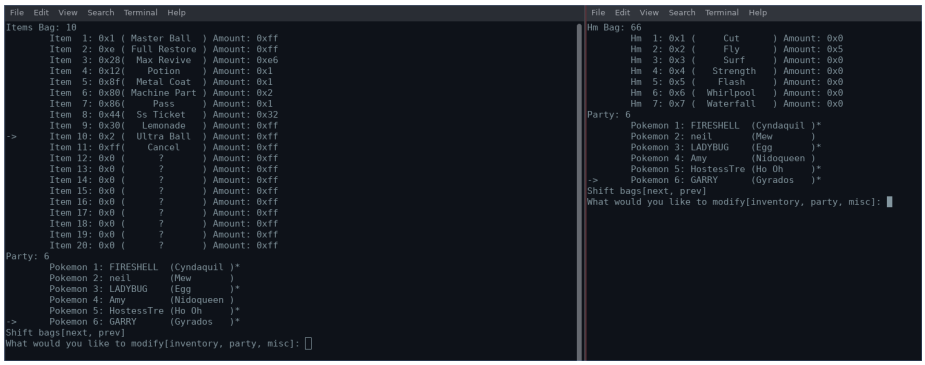


Figure 8: Side by side of the script showing different inventory options

Although the script can actively edit the memory addresses and let you max all of the stats on Pokemon, there are persistent issues with functionality. The hidden stats for the Pokemon allow you to maintain stat increases even after level 100. As such, there is a function in the game that recalculates a Pokemon's stats after a battle or by taking them out of storage. Thus, if this is not accounted for, the changes made to a Pokemon can be removed after a

single battle. The only way to counter this is to get your Pokemon to level 100[7] and then set the stats.

In the end, changing the starter Pokemon options (one of three hard coded Pokemon offered by the professor at the start of the game) was abandoned. This is because the script already gave users the ability to change the Pokemon in their party, so the feature was deemed redundant.

2.5.2 Results

We mentioned in the introduction that Red sits at the peak of a Mt. Silver as a final test of strength in the game. Staying faithful to the idea of an unbeatable *starter* Pokemon (sorry HostessTre), Figure 9 shows the custom Cyndaquil that was created to fight Red's team. FIRESHELL is a max leveled, shiny Cyndaquil with **0xFFFF** health & stats. FIRESHELL's moves were customized by us to take advantage of Red's weaknesses, which are also mostly outside of Cyndaquil's usual move set.

```
Party: 6
->      Pokemon 1: FIRESHELL  (Cyndaquil )*
                HP:     65535/65535
                Level:  100
                EXP:     1059860/1000000
                Item:   Berry
                Moves:
                        Move 1:    thunderbolt( 57)
                        Move 2:  dynamic punch( 57)
                        Move 3:     rock slide( 57)
                        Move 4:   flamethrower( 15)
                Attack (EV):            65535(65535)
                Defense (EV):           65535(65535)
                Speed (EV):             65535(65535)
                Special Attack (EV):    65535(65535)
                Special Defense (EV):   65535(65535)
                Attack/Defense IV:        234
                Speed/Special IV:         170
[pokemon, name, hp, level, exp, item, move #, attack, attEV, defense, defEV, spe
ed, spdEV, spAttack, spDefense, spcEV, attdefIV, spdspcIV]
What would you like to modify:
```

Figure 9: Custom Cyndaquil that defeated Red

The battle was not very eventful as FIRESHELL one-shot the enemies as they spawned. (This means that FIRESHELL could take out each enemy in a single move.) On the second attempt, however, the healing berry that FIRESHELL was carrying was triggered, causing an unnecessary healing sequence down to 9 health even though FIRESHELL had not actually taken any damage. FIRESHELL then missed an attack and had to be revived. Moral of the story: while it is possible to fully outfit your Pokemon for maximum damage and resistance, there are still a lot of game mechanics under the hood that

[7]The actual experience value changes between Pokemon, so guess and check at level 99 may be the most effective way

would need to be found out and accounted for to make our Pokemon unkillable in all scenarios. (In this particular instance, it would have behooved us not to carry heal items into battle.)

Despite these occasional losses, we think the goal of creating an unbeatable starter Pokemon was achieved and can be duplicated by another player, so long as they maintain careful inventory.

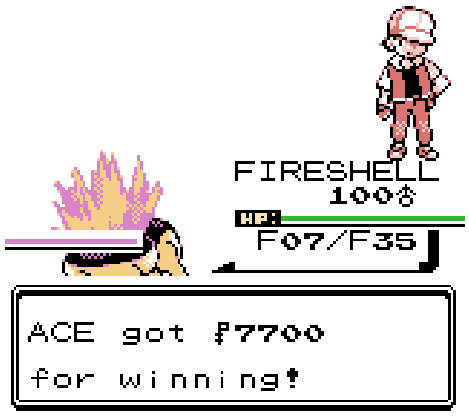


Figure 10: The Unbeatable Starter Pokemon

3 Future Work

This was a very fun project to work on, and we would love to develop it beyond what was required for the class. As stated in the Development section, the framework to implement the Game Boy saves is in place. Given the time limitations, however, we could not analyze both the BGB state and the save. Some work has already been started, but properly documenting the save file for others to use beyond this project is proving a challenge. To help with this, we've looked at a few other software options: Kaitai Struct creates C-like structs and it has a Binja plugin for integration. We are trying to map the save state using Kaitai, but it is essentially learning a new language, which will take time to understand.

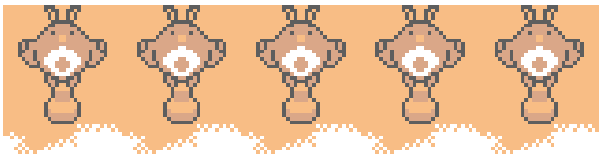
Another idea would be to implement editing the Pokedex, which tracks Pokemon we've seen in the game. The main issue with implementing this feature is that none of our states of the game ever received the device. Once we get our Pokemon, we immediately used the editor to head for Mt. Silver or Kanto.

The ability to manipulate egg data (for hatching additional/specific Pokemon) was suggested by someone too far into development to be implemented before project submission. The idea is interesting, but could be redundant considering the player can give themselves any Pokemon already.

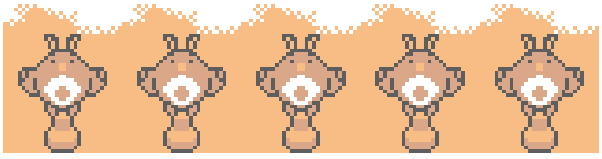
Another idea suggested too late was changing the ROM data in order to

create a custom Pokemon with different types or move sets than those assigned to or associated with that specific Pokemon. We mentioned that there were two values that control the Pokemon: one in the Party struct and the other in the Stats struct. By manipulating these varaibles, we could change how the Pokemon behaves while still displaying the user's favorite Pokemon sprite. However, this goes against the conceit of the project which is creating an unbeatable version of a given Pokemon; to give one Pokemon another Pokemon's moves, we would have to overwrite parts of the ROM in order to save the Pokemon for future use.

As such, we are satisfied with the developed script as it is, both in terms of the power with which the user can imbue their favorite Pokemon, and the user-friendliness of the interface that gives the user so many customizable ways to make their Pokemon unbeatable. There might be a bit of a learning curve, especially when it comes to balancing the desire to destroy an enemy in a single move and the game's automatic resetting of stats after battles. But this script can provide the experienced player with the rare opportunity to play a familiar game in a new and engaging way.



THE END



4 References

Works Cited

BGB Emulator Debugger. 2024. bgb.bircd.org/.

Bulbapedia. 2024. bulbapedia.bulbagarden.net/wiki/Pok%C3%A9mon_data_structure_(Generation_II).

Internet Archive. 2021. archive.org/download/gameboy-color-romset-ultra-us/Pokemon%20-%20Silver%20Version%20%28USA%2C%20Europe%29%20%28SGB%20Enhanced%29%20%28GB%20Compatible%29.zip.

Kaitai Struct. 2024. kaitai.io/.

MisterIcing, phaserphyre. Github. 2024. github.com/MisterIcing/PokemonSilverSaveEditor.

Vector35. Binary Ninja. 2024. binary.ninja/.