15:02 Pier Solar and the Great Reverser

by Brandon L. Wilson

Hello everyone!

I'm here to talk about dumping the ROM from one of the most secure Sega Genesis game ever created.

This is a story about the unusual, or even crazy techniques used in reverse engineering a strange target. It demonstrates that if you want to do something, you don't have to be the best or the most qualified person to do it—you should do what you know how to do, whatever that is, and keep at it until it works, and eventually it will pay off.

First, a little background on the environment we're talking about here. For those who don't know, the Sega Genesis is a cartridge-based, 16-bit game console made by Sega and released in the US in 1989. In Europe and Japan, it was known as the Sega Mega Drive.

As you may or may not know, there were three different versions of the Genesis. The Model 1 Genesis is on the left of Figure 1. Some versions of this model have an extension port, which is actually just a third controller port. It was originally intended for a modem add-on, which was later scrapped.

pierSolar
and the great ~~architects~~ reverser

SEGA GENESIS

Some versions of the Model 1 (and all of the Model 2 devices) started to include a cartridge protection mechanism called the TMSS, or TradeMark Security System. Basically this was just some extra logic to lock up some of the internal Genesis hardware if the word "SEGA" didn't appear at a certain location in the ROM and if the ASCII bytes representing "`S`", "`E`", "`G`", "`A`" weren't written to a certain hardware register. Theoretically only people with official Sega documentation would know to put this code in their games, thereby preventing unlicensed games, but that of course didn't last long.

And then there's the Model 3 of my childhood living room, which generally sucked. It doesn't support the Sega CD, Game Genie, or any other interesting accessories.

There was also a not-as-well-known CD add-on for the Genesis called the Sega CD, or the Mega CD in Europe and Japan, released in 1992. It allowed for slightly-nicer-looking CD-based games as an attempt to extend the Genesis' life, but like many other attempts to do so, that didn't really work out.

Sega CD has its own BIOS and Motorola 68k processor, which gets executed if you don't have a cartridge in the main slot on top. That way you can still play all your old Genesis games, but if you didn't have one of those games inserted, it would boot off the Sega CD BIOS and then whatever CD you inserted.

There were two versions of it, the first one was shaped to fit the Model 1 Genesis, and while the second was modeled for the shape of the Model 2 Genesis, although either would work on the other Genesis. The Model 1 is rare and prone to failure, so it's much more difficult to find. I have the Model 2.

So finally we get to the game itself, a game called Pier Solar. It was released in 2010 and is a "homebrew" game, which means it was programmed by a bunch of fans of the Genesis, not in any way licensed by Sega. Rather than just playing it in an emulator, they took the time to produce an actual plastic cartridge just like real games, make the plastic case for it, nice printed manual, everything just as if it

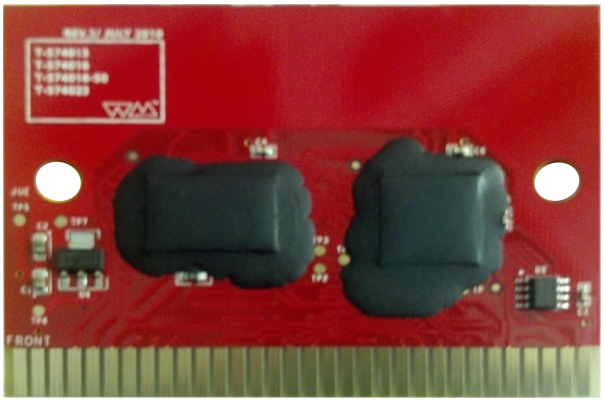
were a real game.

It's unique in that it is the only game ever to use the Sega CD add-on for an enhanced soundtrack while you're playing the game, and it has what they refer to as a "high-density" cartridge, which means it has an 8MB ROM, larger than any Genesis game ever made.

It's also unique in that its ROM had never been successfully dumped by anyone, preventing folks from playing it on an emulator. The lack of a ROM dump was not from lack of trying, however.

Taking apart the cartridge, you can see that they're very, very protective of something. They put some sort of black epoxy over the most interesting parts of the board, to prevent analysis or direct dumping of what is almost certainly flash memory.

Since they want to protect this, it's our obligation to try and understand what it is and, if necessary, defeat it. I can't help it; I see something that someone put a lot of effort into protecting, and I just *have* to un-do it.



I have no idea how to get that crud off, and I have to assume that since they put it on there, it's not easy to remove. We have to keep in mind, this game and protection were created by people with a long history of disassembling Genesis ROMs, writing Genesis emulators, and bypassing older forms of copy protection that were used on clones and pirate cartridges. They know what people are likely to try in order to dump it and what would keep it secure for a long time.

So we're going to have to get creative to dump this ROM.

There are two methods of dumping Sega Genesis ROMs. The first would be to use a device dedicated to that purpose, such as the Retrode. Essentially it pretends to be a Sega Genesis and retrieves each byte of the ROM in order until it has it all.

Unfortunately, when other people applied this to the 8MB Pier Solar, they reported that it just produces the same 32KB over and over again. That's obviously not right, so they must have some hardware under that black crud that ensures it's actually running in a Sega Genesis.

So, we turn to the other main method of dumping Genesis ROMs, which involves running a program on the Genesis itself to read the inserted cartridge's data and output it through one of the controller ports, which as I mentioned before is actually just a serial port. The people with the ability to do this also reported the same 32KB mirrored over and over again, so that doesn't work either.

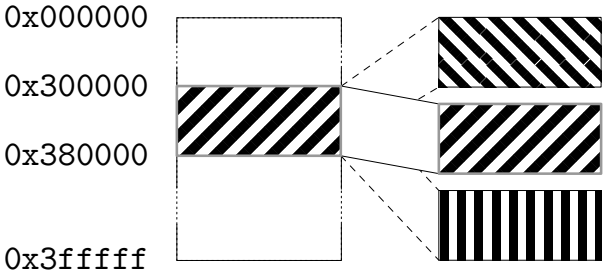
Where's the rest of the ROM data? Well, let's take a step back and think about how this works. When we do a little Googling, we find that "large" ROMs are not a new thing on the Genesis. Plenty of games would resort to tricks to access more data than the Genesis could normally.

Figure 1. From left to right, Sega Genesis models 1, 2, and 3.

The system only maps four megabytes of cartridge memory, probably because Sega figured, "-Four megs is enough ROM for anybody!" So it's impossible for it to directly reference memory beyond this region. However some games, such as Super Street Fighter 2, are actually larger than that. That game in particular is five megabytes.

They get access to the rest of the ROM by using a really old trick called bank switching. Since they know they can only address 4MB, they just change which 4MB is visible at any one time, using external hardware in the cartridge. That external hardware is called a memory mapper, because it "maps" various sections of the ROM into the addressable area. It's a poor man's MMU.

So the game itself can communicate with the cartridge and tell the mapper "Hey, I need access to part of that last megabyte. Put it at address `0x300000` for me." When you access the data at `0x300000`, you're really accessing the data at, say, `0x400000`, which would normally be just outside of the addressable range.



[1] `unzip pocorgtfo15.pdf comcable11.zip`

All this is documented online, of course. I found it by Googling about Genesis homebrew and programming your own games.

So where does this memory mapper live? It's in the game cartridge itself. Since the game runs from the Genesis CPU, it needs a way to communicate with the cartridge to tell it what memory to map and where.

All Genesis I/O is memory-mapped, meaning that when you read from or write to a specific memory address, something happens externally. When you write to addresses `0xA130F3` through `0xA130FF`, the cartridge hardware can detect that and take some kind of action. So for Super Street Fighter 2, those addresses are tied to the memory mapper hardware, which swaps in blocks of memory as needed by the game.

Pier Solar does the same thing, right? Not exactly; loading up the first 32KB in IDA Pro reveals no reads or writes here, nor to anywhere else in the `0xA130xx` range for that matter. So now what?

Well, and this is something important that we have to keep in mind, if the game's code can access all the ROM data, then so can our code. Right? If they can do it, we can do it.

– — — – — — — – — – —

So the question becomes, how do we run code on a Sega Genesis? The same way others tried dumping the ROM—through what's called the Sega CD transfer cable. This is an easy-to-make cable linking a PC's parallel port with one of the Genesis' controller ports, which as I said before is just a serial port. There are no resistors, capacitors, or anything like that. It's literally just the parallel port connector, a cut-up controller cable, and the wire between them. The cable pinout and related software are publicly available online.[1]

As I mentioned before, while the Sega CD is attached, the Genesis boots from the top cartridge slot *only if* a game is inserted. Otherwise, it uses the BIOS to boot from the CD.

Since they weren't too concerned with CD piracy way back in 1992, there is no protection at all against simply burning a CD and booting it. We burn a CD with a publicly-available ISO of a Sega CD program that waits to receive a payload of code to execute from a PC via the transfer cable. That gives us a way of writing code on a PC, transferring it to a Sega Genesis + Sega CD, running it, and communicating back and forth with a PC. We now

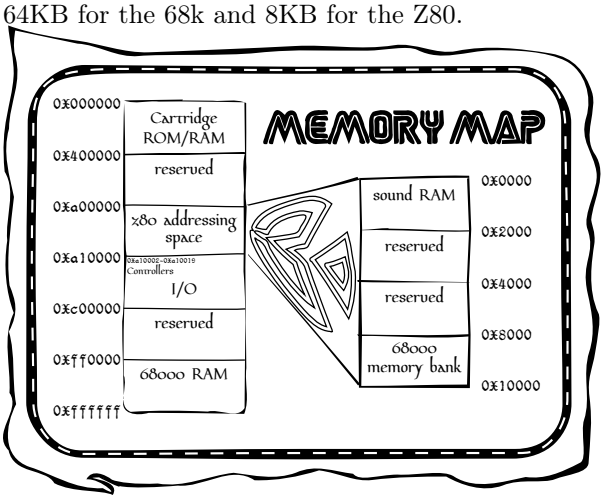
have ourselves a framework for dumping the ROM.

Great, we found some documentation online about how to send code to a Genesis and execute it, now what?

Well, let's start with trying to understand what code for this thing would even look like. Wikipedia tells us that it has two processors. The main processor is a Motorola 68000 CPU running at 7.6MHz, and which can directly access the other CPU's RAM.

The second CPU is a Zilog Z80 running at 4MHz, whose sole purpose is to drive the Yamaha YM2612 FM sound chip. The Z80 has its own RAM, which can be reset or controlled by the main Motorola 68000. It also has the ability to access cartridge ROM—so typically a game would play sound by transferring over to the Z80's RAM a small program that reads sound data from the cartridge and dumps it to the Yamaha sound chip. So when the game wanted to play a sound, the Motorola 68k would reset the Z80 CPU, which would start executing the Z80 program and playing the sound.

So anyway, combined that's 72KB of RAM: 64KB for the 68k and 8KB for the Z80.



Documentation also tells us the memory map of the Genesis. The first part we've already covered, that we can access up to `0x400000`, or 4MB, of the cartridge memory. The next useful area starts at `0xA00000`, which is where you would read from or write to the Z80's RAM.

After that is the most important area, starting at `0xA10000`, which is where all the Genesis hardware is controlled. Here we find the registers for manipulating the two controller ports, and the area I mentioned earlier about communicating directly with the hardware in the cartridge.

We also have 64KB of Motorola 68k RAM, starting at address `0xFF0000`. This should give you an idea of what code would look like, essentially reading from and writing to a series of memory mapped I/O registers.

Reports online are that the standard Sega CD transfer cable ROM dumping method doesn't work, but since we have the source code to it, let's go ahead and try it ourselves. To do that, I needed an older Genesis and Sega CD. I went to a flea market and picked up a Model 1 Sega Genesis and Model 2 Sega CD for a few dollars, then soldered together a transfer cable.

We now have the Sega Genesis attached to the Sega CD and our boot CD inserted, we then cover up the "cartridge detect" pin with tape, so that it won't detect an inserted cartridge. It will boot to the Sega CD.

As the system turns on, the Sega CD and then our burned boot CD starts up. Then the ROM dumping program is transferred over from the PC and executed on the Genesis.

The dump is transferred back to the PC via the transfer cable. We take a look at it in a hex editor, but the infernal thing is *still* mirrored.

Why is this happening? Well, we're reading the data off the cartridge using the Genesis CPU, the same way the game runs, so maybe the cartridge hardware requires a certain series of instructions to execute first? I mean, a certain set of values might need to be written to a certain address, or a certain address might need to be read.

If that's the case, maybe we should let the game boot as much as possible before we try the dump. But, if the game has booted, we're going to need to steal control away from it, which means we need to change how it runs.

Enter the Game Genie, which you might remember from when you were a kid. You'd plug your game into the cartridge slot on top of the Game Genie, then put that in your Genesis, turn it on, flip through a code book and enter your cheat codes, then hit START and cheat to your heart's content.

As it turns out, this thing is actually very useful. What it really does is patch the game by intercepting attempts to read cartridge ROM, changing them before they make it to the console for execution. The codes are actually address/value pairs. For example, if there's a check in a game to jump to a "you're dead" subroutine when your health is at zero, you could simply NOP out that Motorola 68k assembly instruction. It will never take that jump, and your character will never die.

Those of you who grow up with this thing might remember that some games had a "master" code that was required before any other codes. That code was for defeating the ROM checksum check that the game does to make sure it hasn't been tampered with. So once you entered the master code, you could make all the changes you wanted.

Since the code format is documented,[2] we can easily make a Game Genie code that will change the value at a certain address to whatever we specify. We can make minor changes to the game's code while it runs.

Due to the way the Motorola 68k works, we can only change one 16-bit word at a time, never just a single byte. No big deal, but keep it in mind because it limits the changes that we can make.

Well, that's nice in theory, but can it really work with this game? First we fire up the game with the Game Genie plugged in, but *don't* enter any codes, just to see if the cartridge works while it's attached.

Yes, it does, so next we fire up the game, again with the Game Genie plugged in, but *this* time we enter a code that, say, locks up hard. Now, that's not the best test in the world, since the code could be doing something we don't understand, but if the game suddenly won't boot, we know at least we've made an impact.

Now, according to online documentation, the format of a Genesis ROM begins with a 256-byte interrupt vector table of the Motorola 68k,followed by a 256-byte area holding all sorts of information about the ROM, such as the name of the game, the author, the ROM checksum, etc. Then finally the game's machine code begins at address `0x0200`.

If we make a couple of Game Genie codes that place the Motorola 68k instruction "`jmp 0x0200`" at `0x200`, the game will begin with an infinite loop. I tried it, and that's exactly what happened. We can lock the game up, and that's a pretty strong indication that this technique might work.

Getting back to our theory: if the game needs to execute a special set of instructions to make the 32KB mirroring stop, we need to let it run and then take back control and dump the ROM. How do we know when and where to do that? We fire up a disassembler and take a look.

```
1  0x0ec6   2079000015de   movea.l 0x15de.l, a0
   0x0ecc   317c0001000a   move.w 0x1, 0xa(a0)
3  0x0ed2   588f           addq.l 0x4, a7
   0x0ed4   600c           bra.b 0xee2
5  0x0ed6   2079000015de   movea.l 0x15de.l, a0
   0x0edc   317c0001000a   move.w 0x1, 0xa(a0)
7  0x0ee2   0839000000c0   btst.b 0x0, 0xc00005.l
   0x0eea   670e           beq.b 0xefa
9  0x0eec   2079000015de   movea.l 0x15de.l, a0
   0x0ef2   317c0bb80004   move.w 0xbb8, 0x4(a0)
11 0x0ef8   600c           bra.b 0xf06
   0x0efa   2079000015de   movea.l 0x15de.l, a0
13 0x0f00   317c0e100004   move.w 0xe10, 0x4(a0)
   0x0f06   2079000015de   movea.l 0x15de.l, a0
15 0x0f0c   0c680001000a   cmpi.w 0x1, 0xa(a0)
   0x0f12   6608           bne.b 0xf1c
17 0x0f14   4ef90000e000   jmp 0xe000.l
```

[2] `unzip pocorgtfo15.pdf MakingGenesisGGcodes.txt AdvancedGenGGtips.txt`

It is at `0x000F14` that the code takes its first jump outside of the first 32KB, to address `0x00E000`. So assuming this code executes properly, we know that at the moment the game takes that jump, the mirroring is no longer occurring. That's the safest moment to take control. We don't yet have any idea what happens once it jumps there, as this first 32KB is all we have to study and work with.

So we can make 16-bit changes to the game's code as it runs via the Game Genie, and separately, we can run code on the Genesis and access at least part of the cartridge's ROM via the Sega CD. What we really need is a way to combine the two techniques.

So then I had an idea: What if we booted the Sega CD and wrote some 68k code to embed a ROM dumper at the end of 68k RAM, then insert the Game Genie and game while the system is on, then hit the RESET button on the console, which just resets the main 68k CPU, which means our ROM dumper at the end of 68k RAM is still there It should then go to boot the Game Genie this time instead of the Sega CD, since there's now a cartridge in the slot, then enter Game Genie codes to make the game jump straight into 68k RAM, then boot the game, giving us control?

That's quite a mouthful, so let's go over it one more time.

- We write some 68k shellcode to read the ROM data and push it out the controller port back to the PC.
- To run this code, we boot the Sega CD, which receives and executes a payload from the PC.
- This payload copies our ROM dumping code to the end of 68k RAM, which the 32KB dump doesn't seem to use.
- We insert our Game Genie and game into the Genesis. This makes the system lock up, but that's not necessarily a bad thing, as we're about to reset anyway.
- We hit the RESET button on the console. The Genesis starts to boot, detects the Game Genie and game cartridge so it boots from those instead of the CD.
- We enter our Game Genie codes for the game to jump into 68k RAM and hit START to start the game, aaaand...
- Attempting this technique, the system locks up just as we should be jumping into the payload left in RAM. But why?

I went over this over and over and over in my head, trying to figure out what's wrong. Can you see what's wrong with this logic?

Yeah, so, I failed to take into account anything the Game Genie might be doing to mess with our embedded ROM dumping code in the 68K's RAM. When you disassemble the Game Genie's ROM, you find that one of the first things it does is wipe out all of the 68K's RAM.

```
1 0x0294   41f900ff0000    lea.l 0xff0000.l, a0
  0x029a   323c7fff        move.w 0x7fff, d1
3 0x029e   7000            moveq 0x0, d0
  0x02a0   30c0            move.w d0, (a0)+
5 0x02a2   51c9fffc        dbra d1, 0x2a0
```

We can't leave code in main CPU RAM across a reboot because of the very same Game Genie that lets us patch the ROM to jump into our shellcode. So what do we do?

We know we can't rely on our code still being in 68k RAM by the time the game boots, but we need something, anything to persist after we reset the console. Well, what about Z80's RAM?

Studying the Game Genie ROM reveals that it puts a small Z80 sound program in Z80 RAM, for playing the code entry sound effects, like when you're selecting or deleting a character. This program is rather small, and the Game Genie doesn't wipe out all of Z80 RAM first. It just copies this little program, leaving the rest alone.

So instead of putting our code at the end of 68K RAM, we can instead put it at the end of Z80 RAM, along with a little Z80 code to copy it back into 68k RAM. We can make a sequence of Game Genie codes that patches Pier Solar's Z80 program to jump right to the end of Z80 RAM, where our Z80 code will be waiting. We'll then be free to copy our 68k code back into 68k RAM, hopefully before the Game Genie makes the 68k jump there.

```
ROM:0000083A                movem.l d0-d2/a0-a1,-(sp)
ROM:0000083E                move.w  #$100,($A11100).l
ROM:00000846                move.w  #$14,d2
ROM:0000084A
ROM:0000084A loc_84A:                               ; CODE XREF: sub_83A+20↓j
ROM:0000084A                subq.w  #1,d2
ROM:0000084C                beq.w   loc_88A
ROM:00000850                move.w  ($A11100).l,d1
ROM:00000856                btst    #8,d1
ROM:0000085A                bne.s   loc_84A
ROM:0000085C                lea     (unk_19BC).w,a0
ROM:00000860                lea     ($A00000).l,a1
ROM:00000866                move.w  (word_1B1A).w,d0
ROM:0000086A
ROM:0000086A loc_86A:                               ; CODE XREF: sub_83A+32↓j
ROM:0000086A                move.b  (a0)+,(a1)+
ROM:0000086C                dbf     d0,loc_86A
ROM:00000870                move.w  #0,($A11200).l
ROM:00000878                move.w  #0,($A11100).l
ROM:00000880                mulu.w  d1,d0
ROM:00000882                move.w  #$100,($A11200).l
ROM:0000088A
ROM:0000088A loc_88A:                               ; CODE XREF: sub_83A+12↑j
ROM:0000088A                movem.l (sp)+,d0-d2/a0-a1
ROM:0000088E                rts
ROM:0000088E ; End of function sub_83A
```

With this new arrangement, we get control of the 68K CPU *after* the game has booted! But the extracted data is still mirrored, even though we are executing the same way the real game runs.

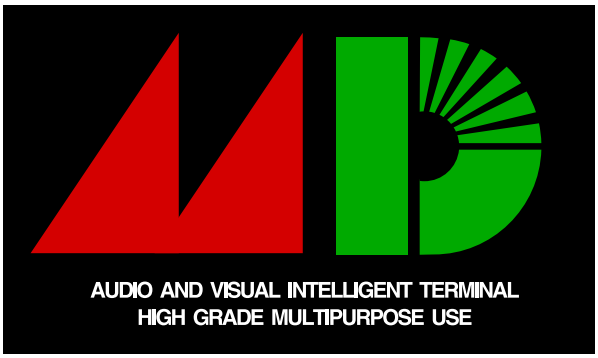
Okay, so what are the differences between the game's code and our code?

We're using a Game Genie, maybe the game detects that? This is unlikely, as the game boots fine with it attached. If it had a problem with the Game Genie, you'd think it wouldn't work at all.

Well, we're running from RAM, and the game is running from ROM. Perhaps the cartridge can distinguish between instruction fetches of code running from ROM and the data fetches that occur when code is running from RAM?

Our only ability to change the code in ROM comes from the Game Genie, which is limited to five codes. A dumper just needs to write bytes in order to `0xA1000F`, the Controller 2 UART Transmit Buffer, but code to do that won't fit in five codes.

Luckily there is a cheat device called the Pro Action Replay 2 which supports 99 codes. These are extremely rare and were never sold in the States, but I was able to buy one through eBay. Unfortunately, the game doesn't boot with it at all, even with no codes. It just sits at a black screen, even though the Action Replay works fine with other cartridges.



So now what? Well, we think that the CPU must be actively running from ROM, but except for minor patches with the Game Genie, we know our code can only run from RAM. Is there any way we can do both? Well, as it turns out, we already have the answer.

We have two processors, and we were already using both of them! We can use the Game Genie to make the 68k spin its wheels in an infinite loop in ROM, just like the very first thing we tried with it, while we use the other processor to dump it.

We were overthinking the first (and second) attempts to get control away from the game, as there's no reason the 68K *has* to be the one doing the dumping. In fact, having the Z80 do it might be the *only* way to make this work.

So the Z80 dumper does its thing, dumping cartridge data through the Sega CD's transfer cable while the 68K stays locked in an infinite loop, still fetching instructions from cartridge hardware! As far as the cartridge is concerned, the game is running normally.

And *YES*, finally, it works! We study the first 4MB in IDA Pro to see how the bank switching works. As luck would have it, Pier Solar's bank switching is almost exactly the same as Super Street Fighter 2.

Armed with that knowledge, we can modify the dumper to extract the remaining 4MB via bank switching, which I dumped out in sixteen pieces very slowly, through lots and lots and lots of triggering this crazy boot procedure. I mean, I can't tell you how excited I was that this crazy mess actually worked. It was like four o'clock in the morning, and I felt like I was on top of the world. That's why I do this stuff; really, that payoff is so worth it. It's just indescribable.

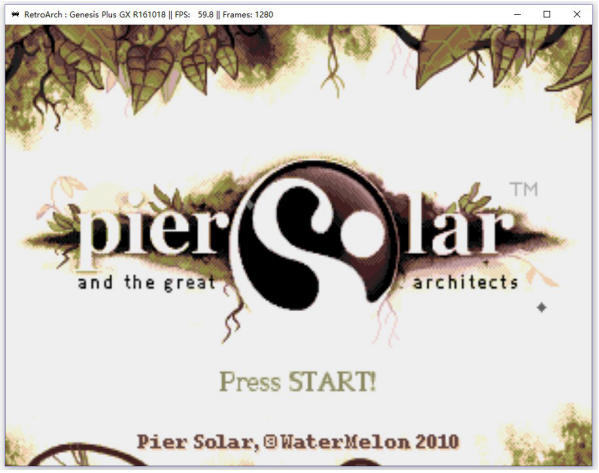
Now that I had a complete dump, I looked for the ROM checksum calculation code and implemented it PC-side, and it actually matched the checksum in the ROM header. Then I knew it was dumped correctly.

Now starts the long process of studying the disassembly to understand all the extra hardware. For example, the save-state hardware is just a serial EEPROM accessed by reads and writes to a couple of registers.

So now that we have all of it, what exactly can we say was the protection? Well, I couldn't tell you how it works at a hardware level other than that it appears to be an FPGA, but, disassembly reveals these secrets from the software side.

The first 32KB is mirrored over and over until specific accesses to `0x18010` occur. The mirroring is automatically re-enabled by hardware if the system isn't executing from ROM for more than some unknown amount of time.

[3]VDP is the display hardware in the Genesis.



The serial EEPROM, while it doesn't require a battery to hold its data, does prevent the game from running in emulators that don't explicitly support it. It also breaks compatibility with those flash cartridges that people use for playing downloaded ROMs on real consoles.

Once I got the ROM dumped, I couldn't help but try to get it working in some kind of emulator, and at the time DGen was the easiest to understand and modify, so I did the bare minimum to get that working. It boots and works for the most part, but it has a few graphical glitches here and there, probably related to VDP internals I don't and will never understand.[3]

Eventually somebody else came along and did it better, with a port to MESS.

Don't think anything is beyond your abilities: use the skills you have, whatever they may be. Me, I do TI graphing calculator programming and reverse engineering as a hobby. The two main processors those calculators use are the Motorola 68K and Zilog Z80, so this project was tailor-made for me. But as far as the hardware behind it, I had no clue; I just had to make some guesses and hope for the best.

"This isn't the most efficient method" and "Nobody else would try this method." are *not* reasons to not work on something. If anything, they're actually reasons *to* do it, because that means nobody else bothered to try it, and you're more likely to be first. Crazy methods work, and I hope this little endeavor has proven that.