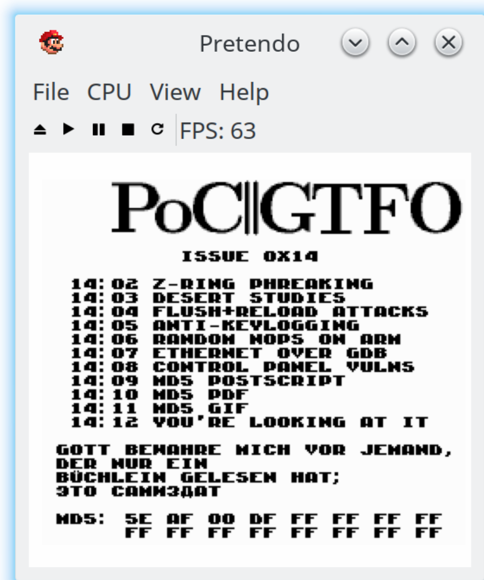
14:12 This PDF is an NES ROM that prints its own MD5 hash!

by Evan Sultanik and Evan Teran

This PDF—in addition to being a ZIP, which is at this point *de rigueur*—is also a Nintendo Entertainment System (NES) ROM that prints out the PDF's MD5 hash. In other words, it is a *hash quine.* The following describes how we did it.



First, we're going to give a quick primer on the NES's hardware architecture, which is necessary to understand the iNES file format, which is ubiquitous for storing ROMs. We then describe the PDF/iNES polyglot, followed by how we achieved the MD5 quine.

NES Hardware and ROMs

NES cartridges have two primary ROM chips: the PRG and CHR. That's one of the reasons why a special file format (*e.g.*, iNES) is necessary to store ROMS: Cartridges don't have a single, contiguous ROM.

The PRG ROM contains the actual executable code of the game. It will typically be loaded into the addresses from `0x8000`–`0xFFFF` of the NES.

We have code, but do we have graphics? That's what the CHR ROM is for![43] The *Picture Processing Unit* (PPU) is what renders the graphics of the NES; it will have either CHR ROM or CHR RAM attached to it. (Note that the PPU has its own address space separate from the CPU.)

Nintendo was clever. Very clever. They knew that the NES console had hardware limitations that developers would inevitably run up against, *e.g.*, the maximum 32 KiB of address space dedicated to the PRG ROM. They allowed cartridges to have custom chips that are able to intercept memory reads (and writes!) and have logic which can effect change based on them. These chips are called *mappers.* That's essentially how the Game Genie works: it is a mapper that sits between the cartridge and the console.

The most basic capability of a mapper is to affect is paging. That's right, around the same time that Intel was releasing the i386, the NES supported basic paging. One common way that this works is that the ROM would detect a write to a ROM at certain addresses, triggering the mapper to switch which pages of ROM were visible where. For example, a cartridge with a NES-UNROM mapper chip would interpret a write of `0x04` to `0x8000` as a command to place the fourth 16 KiB page at address `0x8000`–`0xBFFF`. PRG ROM remapping is just the tip of the iceberg. Mapper hardware grew more and more complex over the years as NES games continued to push the limits of the system.

Mappers are another reason why a ROM format like iNES is required, since there were hundreds of different mapper chips, some specific to individual games. This also makes building an NES emulator very challenging, because each individual mapper chip must be emulated.

The iNES File Format

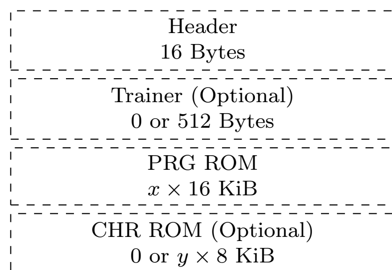
The *de facto* standard for storing NES ROMs is the "iNES format," named after the file format popularized by an early NES emulator by Marat Fayzullin named iNES. While there have been competing file formats over the years such as the "Universal NES Interchange Format" (UNIF), virtually all ROMs you will encounter in the wild will be an iNES file.

It is worth noting that there is a successor to the iNES file format called "NES 2.0." It is backwards compatible with iNES, and adds a few extra types

[43] Or sometimes CHR RAM, as some games procedurally generate their graphics data!

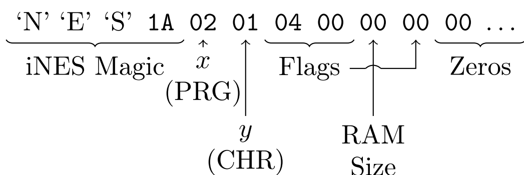
of information, but is not different enough to require discussion for the purpose of creating polyglots. So let's take a look at this format and see where we can place our PDF header safely.

Here is the file format of iNES:



So, what is this strange beast that is a "Trainer"? The trainer section is not something that most ROMs need at all in modern emulators, but any iNES ROM is allowed to have one. Essentially, the trainer is a 512 byte block of code that the emulator will load at memory address `0x7000`–`0x71FF`. Trainers were used by ROM dumpers to store patch code to make it easier to translate commands from an unsupported mapper to one that was supported.

Here is the format of the iNES header:



The third least significant bit of the first flag byte (offset 6) controls whether a trainer section exists. That is why we have set it to `04`.

PDF/iNES Polyglot

As you might have already guessed, the trainer is the perfect place to put our PDF header, since it starts at offset 16 of the iNES file and 512 bytes is more than enough for our PDF header. Ange Albertini first described this approach in PoC||GTFO 7:6. We can then create a PDF object to encapsulate the remainder of the ROM. Since PDF readers ignore everything that comes before the PDF header, the first 16 bytes of the iNES header that come before the Trainer are ignored.

Emulators don't care about data after the ROM data. In fact, you will often find iNES ROMs in the wild that have a URL appended to the end of the file. This causes no harm at all since an iNES file loader only needs to consider the trainer and ROM portions described by the header. Everything afterward—in our case, the remainder of the PDF—is ignored.

So, is it safe to put a PDF header into the trainer? No game which doesn't currently have a trainer will do anything which interacts with code loaded at address `0x7000`–`0x71FF`, so they won't care at all what happens to be there. We had to create our own custom NES ROM to generate the MD5 quine anyway, so we had the control to ensure that the trainer memory was not used.

We fill the trainer with our standard PDF header, containing a PDF object stream to encapsulate the remainder of the NES ROM:

```
%PDF-1.5
%<D0><D4><C5><D8>
9999 0 obj
<<
/Length number of bytes remaining in the ROM
>>
stream
zeros for the remainder of the 512 Trainer bytes
the remainder of the iNES ROM
endstream
endobj
the remainder of the PDF
```

NES MD5 Quine

The next issue is getting the ROM to display its own MD5 hash. We used a technique similar to Greg Kopf's method for a PostScript MD5 quine from article 14:09 up on page 46, however, we were severely restricted by the NES's memory limitations.

In the PostScript MD5 quine PoC, each bit of the MD5 hash was encoded as a two-block MD5 collision that was compared against a copy of itself. That meant that each of the 128 bits of the MD5 hash required four 64 byte MD5 blocks, or 32,768 bytes. That's the size of an *entire* ROM of an NROM-256 cartridge![44] It's twice the amount of ROM that Donkey Kong, Duck Hunt, and Excite Bike required.

We wanted to avoid relying on a mapper. So in order to shrink the hash collision encoding to fit on an NROM-256 cartridge, we only encode one collision (two 64 byte blocks) per MD5 bit. That requires only 16,384 bytes. However, that doesn't al-

[44] NROM-256 is a chip that provides the maximum amount of PRG ROM without using a mapper.

low for the comparison trick that Greg Kopf used in the PostScript quine. One option would be to add a lookup table after the collisions: For each hash collision, encode a diff between the two collided blocks, specifying which block represents "0" and which represents "1". A lookup table would only require an additional 256 bytes (two bytes per MD5 bit). Another option which uses even less space is to take advantage of the fact that Marc Stevens' Fastcoll[45] MD5 collision algorithm produces certain bits that always differ between the two collided blocks, as was described by Kristoffer Janke in article 14:11. So, we can check that bit and use it to determine parity. Either way, after the final PDF is generated and we know its final MD5 hash, we can then swap out each of the collided blocks in the NES ROM to produce the desired bit sequence, all without altering the overall MD5 hash.

This technique requires at most 16,640 bytes of the ROM. However, the MD5 encoding needs to start at the beginning of an MD5 block for the collision to work well (*i.e.*, it needs to start an address that is a multiple of 64 bytes). That means we can't put it at the very end of the PRG ROM, because the last six bytes of that ROM are reserved for the "VECTORS" segment. The NES's CPU expects those six bytes to contain pointers to NMI, reset, and IRQ/BRK interrupt handlers. Therefore, we need to shift the start of the encoding a bit earlier to leave room. In fact, it is to our advantage to have the MD5 encoding occur as early as possible—having as much of our code occur after it as possible—because any changes that occur after the 16,640 bytes of MD5 encoding will *not* require recomputing the hash collisions. Therefore, we chose to store it starting at memory offset `0x9F70`, which corresponds to byte `0x9F70` − `0x8000` = `0x1F70` in the PRG ROM, which corresponds to byte 16 + 512 + `0x1F70` = `0x2180` within this PDF. Feel free to take a gander!

The code in the NES ROM to read the encoded MD5 hash looks something like that in Figure 12.

The music in the ROM is *Danger Streets*, composed and released to the public domain by Shiru, also known as DJ Uranus.[46]

[45] `unzip pocorgtfo14.pdf fastcoll-v1.0.0.5-1.zip`
[46] `https://shiru.untergrund.net/`

```
 1 /* memory address of the start to the encoded MD5:  */
   #define MD5_OFFSET        0x9F70
 3 /* memory address of the lookup table: */
   #define MD5_DIFFS_OFFSET (MD5_OFFSET+128*128) /* 128*128 = 16,384 bytes */
 5 /**
    * Reads one of the 16 bytes from the encoded MD5 hash
 7  */
   uint8_t read_md5_byte(uint8_t byte_index) {
 9     uint8_t byte = 0;
       for(uint8_t bit=0; bit<8; ++bit) {
11         uintptr_t diff_offset =  MD5_DIFFS_OFFSET   /* lookup table encodes the byte   */
                                   + 2 * 8 * byte_index /* index that is different         */
13                                 + 2 * bit);          /* between the collided blocks     */
           uintptr_t offset =  MD5_OFFSET
15                             + 128 * 8 * (uintptr_t)byte_index /* 1024 B per encoded byte */
                               + 128 * (uintptr_t)bit
17                             + PEEK(diff_offset);     /* index of the byte to compare    */
           byte <<= 1;
19         if(PEEK(offset) == PEEK(diff_offset + 1)) {  /* second byte of the lookup table */
               byte |= 1;                               /* encodes the value of the byte   */
21         }                                            /* in the collision block that     */
       }                                                /* represents "1"                  */
23     return byte;
   }
```

Figure 12. Colliding Block Reader