Looking at the RarVM

It has been 7 years now since Tavis Ormandy published his research about the obscure Virtual Machine unrar provided to execute custom code that was able to be attached to RAR archives.
Based on his blog post[1] and the small tool chain[2] he developed I looked into the topic recently as well and thought that it might be something worth resurfacing, especially considering obsolete things like this seem to pop up time and time again[3]:

unrar and WinRAR prior to version 5.00 contained logic for optional filter code meant to optimize the compression for custom data formats.
When extracting files of an archive that made use of it the optional code was parsed and executed by an internal embedded Virtual Machine - the RarVM.

Custom programs inside archives sound fascinating, but this feature was never really used in the creation of archives by the official tools. Because of its lack of outside communication, interacting with it is also not possible, so it doesn't provide other uses either. Regardless it's an interesting example of an embedded VM.

The VM itself provides 8 general purpose 32-bit registers, a program counter, 3 different flags (zero, carry and signed) and a 262 kB data address space for processing the files meant for extraction.
Executable filter code is completely separated from the addressable memory and limited to 25,000,000 executed instructions per filter, but the amount of filters per archive is not restricted.

Although the instruction set isn't special with its 40 different instructions, it not only covers all common arithmetic and logical operation, but also basic stack based instructions and jumps.
Only the PRINT and STANDARD instructions deviate from what could be expected, where the PRINT instruction does nothing in the later versions of the RarVM and was originally used as a debug instruction to dump the state of registers[4].
Contrarily STANDARD is the only instruction actually used in archive creation supported by WinRAR and is responsible for invoking the predefined standard filters which, for example, cover optimised compression for Pentium and Itanium program code.

[1]http://blog.cmpxchg8b.com/2012/09/fun-with-constrained-programming.html
[2]https://github.com/taviso/rarvmtools
[3]https://github.com/pr0cf5/CTF-writeups/tree/master/2019/real-world-ctf-2019
[4]https://github.com/pmachapman/unrar/commit/30566e3abf4c9216858bae3ea6b44f048df8c4a5#diff-9fbcab26b4523426ccb1520539e7408b

An example of how a very basic filter looks like:

```
mov     [#0x00001000], #0x65676150
mov     [#0x00001004], #0x74754F64
mov     [#0x0003C020], #0x00001000
mov     [#0x0003C01C], #0x00000008
jmp     #0x00040000
```

```
./unrar p -inul pagedOut.rar
PagedOut
```

This small filter first moves the "PagedOut" string to address 0x1000, updates the data pointer to that address, then resizes the uncompressed output length to 8 bytes and at the end jumps out of the address space to indicate successful execution.
Regardless of what file data was filtered the output will always stay the same with this filter, although other data could appear before and after it.

Looking deeper into the parsing of the code the instruction encoding is even more interesting as instructions are not even byte aligned while in binary format. The first few bits of an instruction indicate the opcode and can either be 4 or 6 bits in length.
In case the instructions support the "ByteMode" which most that operate on memory do, another bit is added that decides if the operation accesses four or just a single byte at a time. Lastly follows the encoding of the operands, which differ depending on whether they encode a register, an immediate value, or a memory reference. For the immediate values the number to encode decides the bit lengths, and for memory references whether they include an index register, a base address or both.
Notable here is that all instructions with operands support all encodings for all their parameters. This allows for self-modifying code when setting the destination operand to an immediate value:

```
./rarvm-debugger d -trace example02.rar
[0000] SUB      #0x00000002, #0x00000001
[0001] JNZ      #0x00000001
[0000] SUB      #0x00000001, #0x00000001
[0001] JNZ      #0x00000001
```

There is quite a lot more to look into, so if any of this sounded fun I can only recommend looking into the aforementioned blog post and the source code of an unrar version still containing the VM[5]. Additionally I've also collected some information, a small debugger and some example archives as well[6].

[5]https://github.com/pmachapman/unrar/tree/bca1c247dd58da11e500013130a22ca64e830a55
[6]https://github.com/Pusty/rarvm-debugger