Investigating Rust's Anonymous Symbols for Embedded Contexts

Gabe Barney

July 25, 2023

1 The concept of embedded data and why it's important

Embedded data is considered the disk space that is not necessarily allocated by the application, but rather the constructs of the compiler and language that increase the size of the binary. Currently, we detect embedded data by parsing for the space between different symbols which tells us how much padding there is, as well as the space that is used by constant strings.

2 What I've looked at

2.1 Manually attributing symbols to functions on ARM

```
00010694 <_ZN106_...>:
   ...
   106d4:       4805            ldr     r0, [pc, #20]   ; (106ec <_ZN106_...)
   106d6:       2139            movs    r1, #57 ; 0x39
   106d8:       4a05            ldr     r2, [pc, #20]   ; (106f0 <_ZN106_...>)
   106da:       f019 feb5       bl      2a448 <_ZN4core9panicking5panic17ha9e216274a4e0a49E>
   106de:       defe            udf     #254    ; 0xfe
   106e0:       4a01            ldr     r2, [pc, #4]    ; (106e8 <_ZN106_...)
   106e2:       f020 f96b       bl      309bc <OUTLINED_FUNCTION_272>
   106e6:       defe            udf     #254    ; 0xfe
   106e8:       00039718        .word   0x00039718
   106ec:       000315c0        .word   0x000315c0
   106f0:       000396dc        .word   0x000396dc
```

Figure 1: An example of ldr instructions and ARM words

```
000315c0 <str.0>:
   315c0:       7461 6574 706d 2074 6f74 6320 6c61 7563     attempt to calcu
   315d0:       616c 6574 7420 6568 7220 6d65 6961 646e     late the remaind
   315e0:       7265 7720 7469 2068 2061 6964 6976 6f73     er with a diviso
   315f0:       2072 666f 7a20 7265 d46f d4d4 4050 0003     r of zero...P@..
   31600:       0021 0000 02dc 0000 0022 0000 754d 4178     !........"...MuxA
   31610:       5345 3231 4338 4d43 203a 7263 7079 5f74     ES128CCM: crypt_
   31620:       6f64 656e 6920 2073 6163 6c6c 6465 6220     done is called b
   31630:       7475 6920 666e 696c 6867 2074 7369 6e20     ut inflight is n
   31640:       6e6f 2165 6163 7370 6c75 7365 732f 6372     one!capsules/src
   31650:       762f 7269 7574 6c61 615f 7365 635f 6d63     /virtual_aes_ccm
   31660:       722e d473 1644 0003 001f 0000 00e9 0000     .rs.D...........
   31670:       000d 0000 6143 6c6c 6465 7220 6165 5f64     ....Called read_
   31680:       6c62 636f 286b 2029 6977 6874 6e20 206f     block() with no
   31690:       6164 6174 0000 0000 0020 0000 0000 0000     data.... .......
   316a0:       0002 0000 0000 0000 0000 0000 0002 0000     ................
```

Figure 2: An example symbol in ARM's literal pool

In ARM, there are '.word' directives which are a level of indirection that makes parsing symbols directly through pure string manipulation very difficult because of address alignment. Compounding upon this, because words are an arbitrary directive, we can't always tell what the meaning for each of the words for functions mean. This is because there are sometimes words that are generated to be used

for padding in order to align the literals in the literal pools. We can't understand these automatically generated words without looking into each individual case. Another issue is that when we find the location of a word, often times we don't know the length, due to the difficulty of trying to manually parse registers. With some functions, such as basic panics such as in Figure 1, it is easy to understand the length of the literal that's being referred to, but we find that the coverage of this byte-wise isn't nearly as significant as we would hope.

Approximately 52% of the imix board embedded data is panic-related, this is deduced by removing the panic handler and comparing binary sizes. By parsing the length of strings used in panics by detecting the string length registers, which in Figure 1's case is r1, there is a surprisingly low 8% of this panic-related data that is attributed with this method. And this is the majority case of panic data that we can detect manually. It is also by far the simplest.

The main issue that occurs here seems to be rooted in ARM's literal pooling. Parsing strings, like in Figure 2, is difficult when they're in such large collections with unrelated strings immediately surrounding them. We can easily attribute the first string in Figure 2 from Figure 1's assembly through string manipulation, but what about cases that aren't as explicit and straightforward as panics? Additionally, ARM seems to mix in unreferenced embedded strings with referenced ones, which greatly adds to the complexity when we were looking at the imix binary.

It's not a simple task to detect embedded data in ARM, you would expect that looking at the symbol table would easily give you the size of all the embedded string symbols, even if they're clumped together with unrelated strings. But it turns out that the symbol sizes in the symbol table are the size of only the first string in that symbol. This means that any string past the first string is not detected in the size of that symbol, and that you could only really detect the true overall size of a symbol by looking at the difference in the addresses from one symbol to its neighbor.

With ARM, there is limited 4KB offset for the LDR instruction, which can make it understandable that there are some oddities in ARM's literal pool. This gives some inspiration to changes in platform to RISC-V that can be read in Section 2.4.

2.2 llvm-dwarfdump and pyelftools

Quickly realizing that manually parsing lengths of these anonymous symbols is too difficult, especially for such low coverage. Trying to leverage the ELF and DWARF tooling available became attractive, as it would be far more generalized. This would be beneficial because we could get more coverage of the panics, as well as attribute embedded data to more functions beyond just panics.

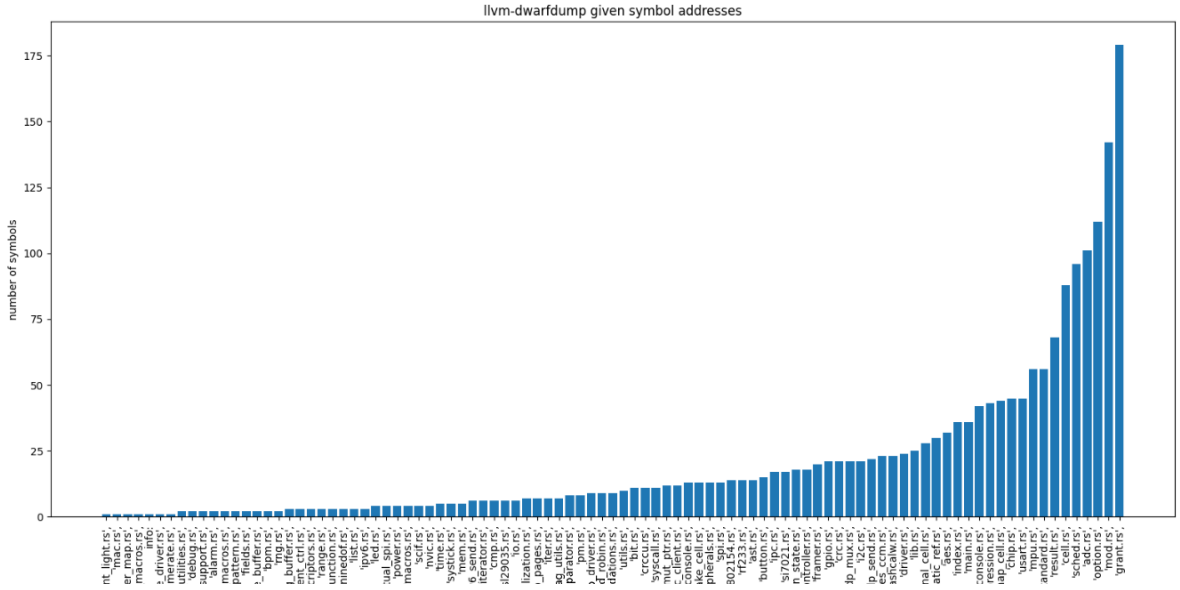

Figure 3: This plot shows how often these tools attribute symbols to files

I believe that pyelftools[1] could be particularly useful for attributing symbols to functions, but its surface value seems to be equivalent of llvm-dwarfdump. LLVM's dwarfdump is simply faster, so I resorted to using it after a brief testing period.

From Figure 3, we can see that the tool doesn't seem to have an inappropriate attribution scheme, and can attribute embedded data to files and line numbers. But the different DWARF outputs from similar embedded data made it obscure to establish further findings, particularly the functions in which they occur. The tools output the line and column numbers of where the symbols should be, but it is still completely uncertain how or where the embedded data comes into play. These locations that are received don't quite output interesting code, and make it seem that this doesn't quite achieve what is wanted. There is potential promise, but it seems to be best left alone for now. Those with extensive DWARF experience could probably figure out more.

2.3 Determining the influence of LLVM's Link Time Optimization

Discussion led to investigating how LTO affects the organization of the binary in ARM. It caused a minor increase in number of symbols, but made no clear improvement in the ability to easily parse and attribute strings to functions, as they were still heavily clumped together with unrelated strings. The importance in this is that this further cements the conclusion that this is most likely an issue with ARM.

2.4 RISC-V analysis

The lack of meaningful information gained from turning off LTO on ARM prompted looking at a RISC-V board. Coming into this without knowing any differences between ARM and RISC-V, it was natural to check if LTO has any influence on RISC-V, then attempting to see how easily symbols can be attributed to functions, like in Section 2.1.

Having LTO off has 779 symbols related to embedded data while having LTO on has 675 symbols related to embedded data. This makes intuitive sense.

In RISC-V, there is a new symbol type. It's titled L__unnamed, yet its functionality seems identical to Lanon symbols. Yet, turning LTO on causes there to be more Lanon than unnamed, and turning LTO off causes there to be more unnamed than Lanon. It seems there's no functional difference between these two different symbol types, but their change in distribution with LTO turned on/off is interesting. Additionally, there seems to be a range, increasing from 170, in the __unnamed_ symbol naming scheme that corresponds to them being unreferenced.

```
20011cc0 <str.0>:
20011cc0:       7461 6574 706d 2074 6f74 6320 6c61 7563     attempt to calcu
20011cd0:       616c 6574 7420 6568 7220 6d65 6961 646e     late the remaind
20011ce0:       7265 7720 7469 2068 2061 6964 6976 6f73     er with a diviso
20011cf0:       2072 666f 7a20 7265                          r of zero
```

Figure 4: An example symbol in RISC-V's literal pool

It's reasonable to believe that the limitation on LDR offsets, is a potential reason for the literal pool issues, and is therefore the reason why ARM is so hard to parse. There is no behavior alike this limitation on LDRs in RISC-V. When looking at symbols in RISC-V that were clumped together in ARM, they are completely isolated to themselves.

This property of RISC-V, along with the removal of the words from ARM, make it very easy to parse through the assembly and attribute the symbols to functions. Making attributions and code size contributions for functions simple. From the limited information visible in Figure 5, I've completed a mapping of functions to their embedded string byte usages. This also means t hat it's possible to know the mapping of all the symbols to their functions they're used in.

Given the long names of functions as well as the difficulty in representing the y-axis, if you want to know the details of this plot, as well as other related plots, you'll need to hover over the bar by graphs using the script or the notebook. [2]

[1] https://github.com/eliben/pyelftools
[2] https://github.com/barneyga/tock-embedded-data

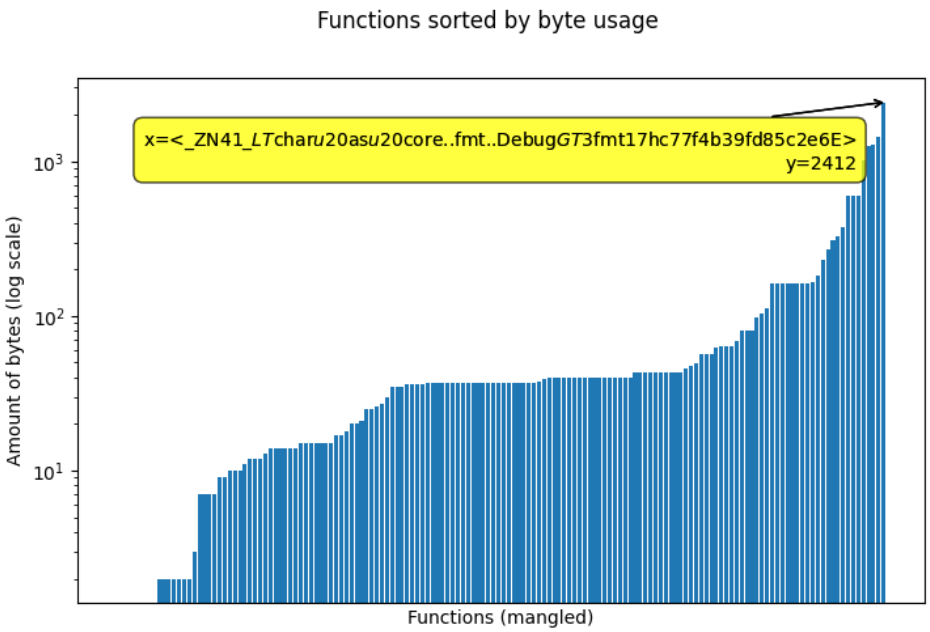

Figure 5: A plot showing the embedded string byte contribution per function

2.5 Next steps

1. Embedded data detection doesn't work the same for RISC-V boards as it does with ARM board. This makes it hard to comprehend the coverage we get from mapping the symbols to their functions.

2. Following from this, being able to isolate certain functions and determine their coverage and individual attributes, currently I can only think of compiling without the panic handler, as done before, but I'm unsure as to how other features could be isolated.

3. Expanding the current print_tock_memory script with a flag that prints information about the embedded data organized by their parent functions.