White Paper

Separating instructions and data with PureCode

Andre Simoes Dias Vieira

June 2016

1. Introduction

Micro-controllers are important and ubiquitous in the modern world and are becoming more so via the Internet of Things (IoT). Good support for microcontrollers is an essential part of a modern software development toolkit. For reasons of cost, energy efficiency, robustness, performance, and security, micro-controllers often feature on-chip memory. For reasons of trade-off between energy efficiency and performance, many low-cost micro-controllers implement low-power on-chip non-volatile memory which has low access speed. There are additional system-level techniques that enable the designs to achieve good performance. Some of these techniques, such as ash prefetching logic, may favor sequential instruction fetches but do not improve data fetch performance. When working with such a component, the developer may need to separate instructions and data in the program's memory to get the best performance. To support this requirement, ARM® is introducing a new processor-specific ELF section attribute `SHF_ARM_PURECODE`.

The remained of this whitepaper is organized as follows:

- Section 2 describes the definition of the new ELF section attribute PureCode.
- Section 3 contains use case scenarios for possible applications of the separation of instructions and data.
- Section 4 contains small code examples that illustrate the use of the PureCode section attribute throughout compilation and linking.

2. PureCode

The ARM processor-specific ELF PureCode section attribute, `SHF_ARM_PURECODE`, is used to describe a section that contains only instructions and no data. During linking, such sections must be put into segments that are executable. These sections do not require the ability to be read, however it is permitted for a linker to place them into the same segment as other text sections. When the segment only contains sections with the `SHF_ARM_PURECODE` attribute, the only required access permission for that segment is execution. This means that the ELF ABI segment attribute for such segments must be PF_X. As per the System V ABI, the segment attribute PF_X requires that segment to be executable. A system consuming such a segment is permitted to treat it as readable [1].

3. Use Case Scenarios

In this section we will see examples of how splitting instructions and data can be beneficial for both performance and security.

3.1 Performance

The Harvard architecture describes a computer architecture with separate memories for data and instructions. In a pure Harvard architecture the part of a processing element that is responsible for reading instructions, cannot read data, as it only has the physical connections required to read instructions. Similarly, the part of a processing element responsible for reading data can only read data. This can result in a reduction of the number of connections required in the system, which can reduce power consumption. To enable such a design, a software developer would need to maintain full separation of instructions and data. To do this, they would need to forbid literal pools or any other form of data inside text sections.

In certain cases, even when it is possible to read data from instruction-memory, it can be undesirable to do so for performance reasons. Take for example systems for which instruction-memory has been optimized for instruction fetches. Doing so will reduce, or even do away with delays for sequential instruction fetches. However, accessing data residing in instruction-memory will cause clock cycle delays due to flash wait states. In such cases, separating data and instructions leads to better performance.

3.2 Security

The emergence of micro-controllers in the connected world of IoT means these previously isolated devices now have to communicate with other devices, making them vulnerable to outside attacks. In a bare-metal environment, without the protection layers offered by an OS, and in the absence of more advanced defensive mechanisms like ARMv8-M Security Extensions, disabling the ability to read code can disable certain threat models.

Blind Return Oriented Programming (BROP) is a blind hacking method that uses single stack vulnerabilities to read the vulnerable binary and transfer it over the network [2]. This is only possible because the memory holding the instructions has read permissions. Making such memory execute-only makes such attacks impossible, in turn making the device less vulnerable to other forms of attack. To enable memory containing instructions to be execute-only, the sections which contain these instructions must not contain data.

4. Code Examples

In this section we use code examples to illustrate how to separate instructions and data using the PureCode section attribute.

Assume we want to compile the simple C code in Figure 1, such that there is a separation of instructions and data. Figure 2 shows a C code function for which we would like to generate an executable with a separation between instructions and data. By default, the compiled assembly for this code for an ARMv7-M architecture target could make use of a literal pool, containing the 64 bit constant used in the calculation of the argument for `bar()`, as can be observed in Figure 2.

```
extern int bar (long long);

int foo(long long a)
{
  return bar (428902378902378900 + a);
}
```

Figure 1: Simple Example C code.

To achieve separation of instructions and data for such a function, we have to prevent the generation of literal pools inside the section containing the instructions. Instead, the compiler must generate either loads from a data section, or moves with immediates. In Figure 3 we see an example of code using the latter. The section attributes of this section include the numeric value `0x20000000`, which is the value of the SHF_ARM_PURECODE section attribute.

```
    .section .text.foo,"0x6",%progbits
    .align 1
foo:
    adr     r3, .L2
    ldrd    r2, [r3]
    adds    r0, r0, r2
    adcs    r1, r1, r3
    b       bar
.L3:
    .align 3
.L2:
    .word   -1312320062
    .word   9986161
```

Figure 2: ARMv7-M assembly for the simple example code in Figure 1.

After assembly, this translation unit has to be linked with the rest of the program. Since `bar()` is defined in another translation unit, it is only at link time that its address is known. It is possible that the memory location of `bar()` is further away from `foo()` than the maximum allowed range of a branch instruction with an immediate operand. In such cases, the linker must insert a veneer containing a branch instruction with a register operand, which has a wider range. In this veneer, the address of `bar()` is usually loaded from a literal pool into the register used as the branch operand. However, these veneers are placed in the same section as the code that jumps to it, in our case this is a PureCode section. When introducing literal pools into this veneer, the linker breaks the 'no-data' requirement for such sections. To prevent this, the linker must not use literal pools for veneers within such sections. Instead, the linker could use R_ARM_THM_MOVW_ABS_NC and R_ARM_THM_MOVT_ABS relocations [3], which make use of the `MOVW` and `MOVT` instructions to encode the address to jump to, similar to how the constant was encoded in `foo()`. In Figure 4 we see the code generated for such a veneer within a section with the PureCode attribute.

```
    .section .text.foo,"0x20000006",%progbits
    .align 1
foo:
    movw    r2, #38338
    movw    r3, #24689
    movt    r2, 45511
    movt    r3, 152
    adds    r0, r0, r2
    adcs    r1, r1, r3
    b       bar
```

Figure 3: ARMv7-M assembly for the simple example code in Figure 1 with no literal pool.

```
__bar_veneer:
    movw    ip, #33049    ; 0x8119
    movt    ip, #0
    bx      ip
```

Figure 4: ARMv7-M assembly for a veneer within a section with the SHF_ARM_PURECODE attribute.

Bibliography

[1] "System V ABI," 2015. [Online]. Available: http://www.sco.com/developers/gabi.

[2] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres and D. Boneh, "Hacking Blind," *2014 IEEE Symposium on Security and Privacy (SP),* pp. 227-242, 2014.

[3] "ELF for the ARM Architecture (ARM IHI0044F)," ARM, 2015. [Online]. Available: http://infocenter.arm.com.

Trademarks