arm

Dynamic Configuration and PIE for TF-A

Soby Mathew

Tech Lead

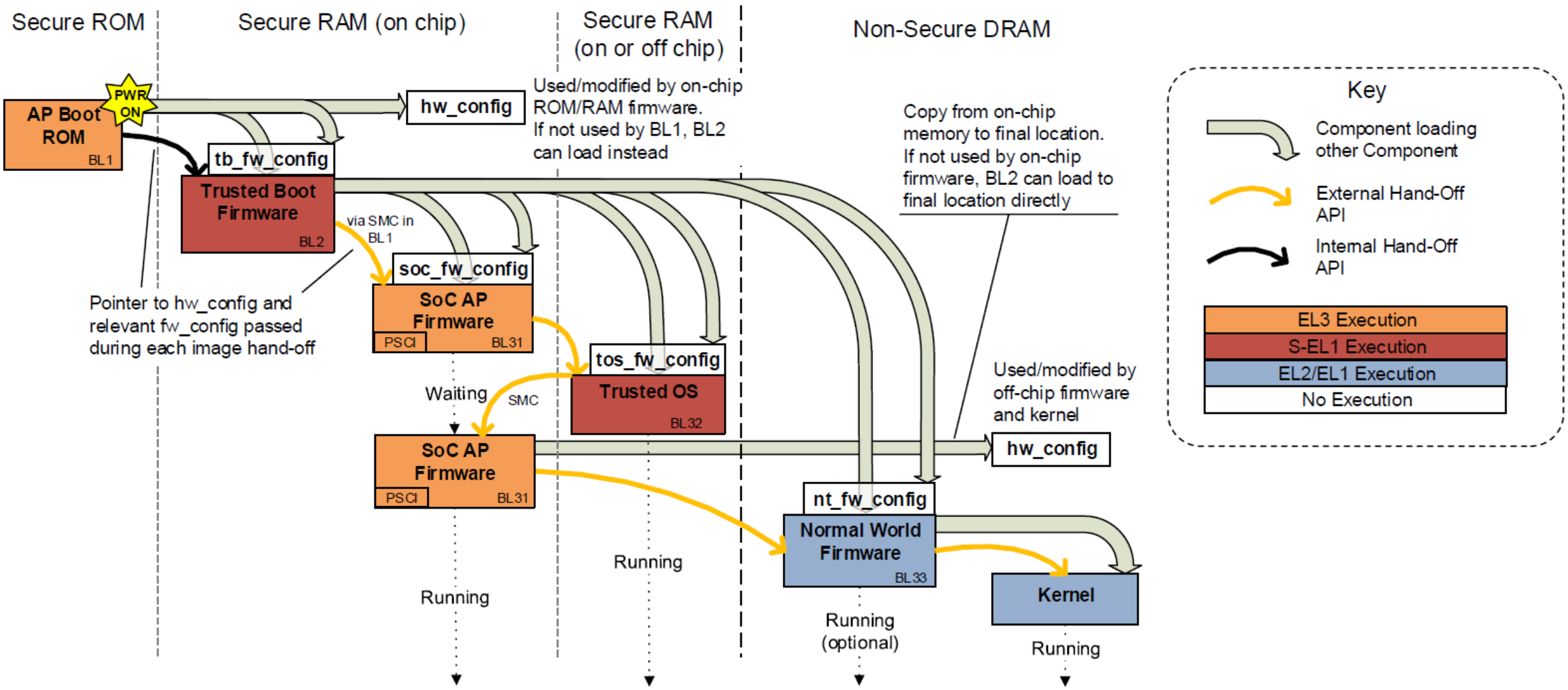
Trusted Firmware-A

05th April 2019

Agenda

- Describe the Dynamic Configuration feature as implemented in TF-A with some common use-cases
- Explain how Position Independent Executable (PIE) Support is enabled in TF-A

Dynamic Configuration Recap



Dynamic Configuration Introduction

- SFO'17 had a BoF session discussing this for firmware.
  - https://connect.linaro.org/resources/sfo17/sfo17-310/
  - This is an optional feature.
  - DTB is the format used for config files on ARM reference platforms
- Some of the envisaged use-cases :
  - Dynamic config of secure firmware features, hardware configuration, security policies
  - Modification of hardware configuration as seen by other software
  - Passing information between BL images
- We will see some example use-cases later.

TF-A implementation details

- Necessary framework in place for most of the use-cases as described in SFO'17 session
  - BL32 (Trusted OS) does not make use of the configuration files yet.
  - Static firmware configuration is better implemented in combination with a config driven build system (like Kbuild)
- Unified argument passing between each BL image
  - Each BL image can now pass 4 registers as arguments.
  - Some of the arguments between BL images are mandated as part of hand-off
  - Other arguments are platform defined. ARM platforms use the following convention:

BL1 -> BL2 arguments	
arg0	tb_fw_config
arg1	Memory info from BL1
arg2	hw_config
arg3	unused

BL2 -> BL31 arguments	
arg0	List of executable images
arg1	soc_fw_config
arg2	hw_config
arg3	Magic number

BL31 -> BL33 arguments	
arg0	nt_fw_config
arg1	hw_config
arg2	unused
arg3	unused

arm

TB_FW_CONFIG example from FVP

hw_config is usually the kernel DTB. It can be accessed by all BL images.

This can turn OFF TBBR authentication dynamically. The TB_FW_CONFIG itself must be authenticated first.

The load addresses of various other config files for following BL images.

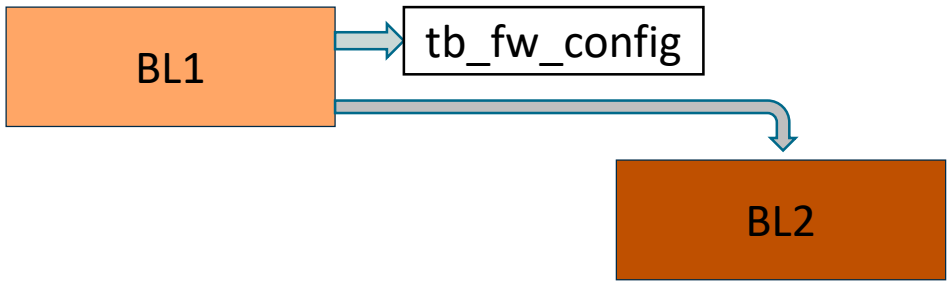
The heap to be shared by BL1 and BL2 for mbedTLS. This is a placeholder which will be populated by BL1 at runtime.

```
/dts-v1/;

/ {
        /* Platform Config */
        plat_arm_bl2 {
                compatible = "arm,tb_fw";
                hw_config_addr = <0x0 0x82000000>;
                hw_config_max_size = <0x01000000>;
                /* Disable authentication for development */
                disable_auth = <0x0>;
                /*
                 * Load SoC and TOS firmware configs at the base of
                 * non shared SRAM. The NT firmware config is loaded in DRAM
                 */
                soc_fw_config_addr = <0x0 0x04001000>;
                soc_fw_config_max_size = <0x200>;
                tos_fw_config_addr = <0x0 0x04001200>;
                tos_fw_config_max_size = <0x200>;
                nt_fw_config_addr = <0x0 0x80000000>;
                nt_fw_config_max_size = <0x200>;

                /*
                 * In case of having shared Mbed TLS heap between BL1 and BL2,
                 * BL1 will populate these two properties with the respective
                 * info about the shared heap. This info will be available for
                 * BL2 in order to locate and re-use the heap.
                 */
                mbedtls_heap_addr = <0x0 0x0>;
                mbedtls_heap_size = <0x0>;
        };
};
```

Usecase-1: Disable TBB for development

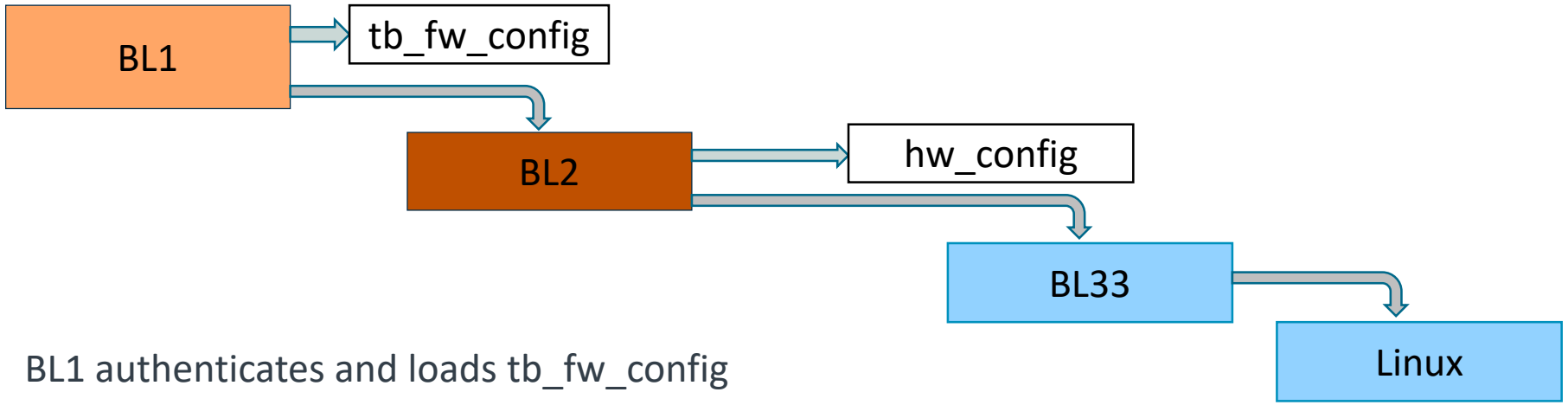


This is an example of dynamic configuration of Firmware features.

This is mainly useful for firmware development. All images except tb_fw_config can be updated without regenerating CoT certificates.

1. BL1 authenticates and loads tb_fw_config
2. It reads whether TBB is to be enabled or not. If disabled, then BL2 is loaded without authentication
3. Similarly BL2 reads tb_fw_config and loads other BL images without authentication if the config specifies to disable TBB

Usecase-2: Load & Authenticate Kernel DTB dynamically

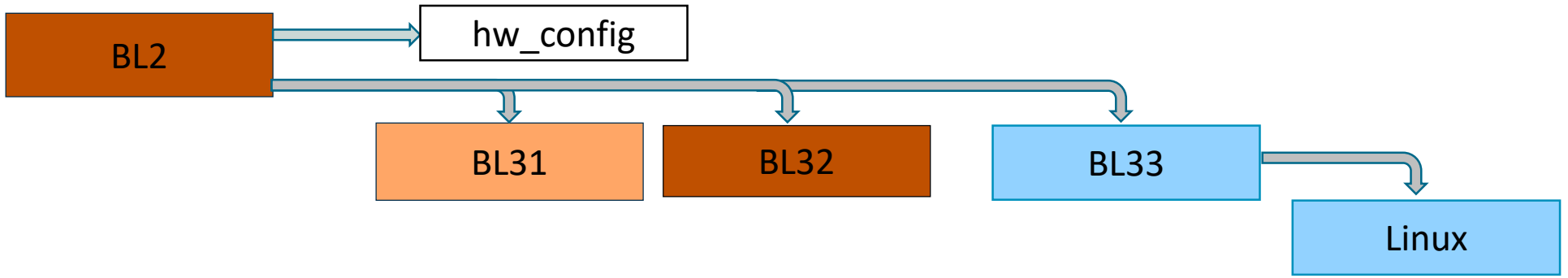


1. BL1 authenticates and loads tb_fw_config
2. BL1 authenticates and loads BL2
3. BL2 reads the tb_fw_config and based on the address specified by it, authenticates and loads hw_config (Linux Kernel DTB).
4. BL2 loads BL33 and configures the right arguments to BL33 to point to hw_config.
5. BL2 hands to the next BL image and finally execution reaches BL33.
6. BL33 loads and hands off to Linux Kernel with x0 pointing to hw_config

Other configs like soc_fw_config and tos_fw_config are optional and can be loaded based on description in the tb_fw_config

arm

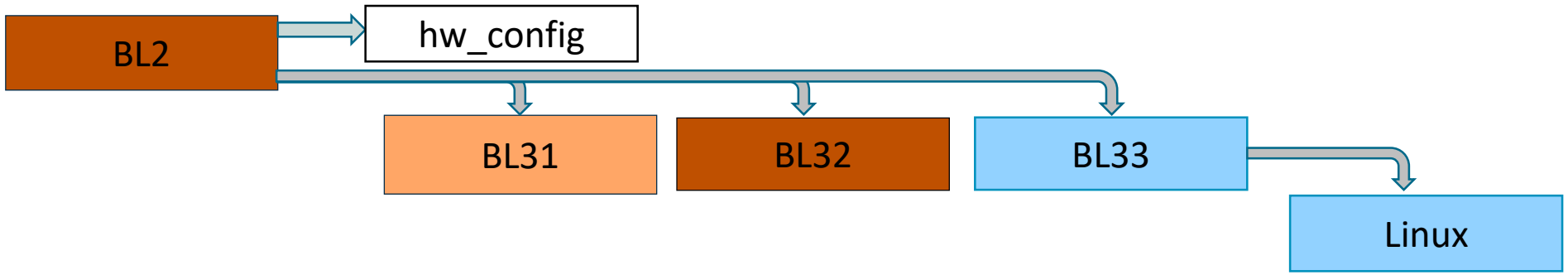
Usecase-2a: Use device description in hw_config



This is an example of dynamic firmware configuration using hw_config.

1. The BL images use device information in hw_config to configure them. (Eg: GIC).
2. Secure devices can also be described in hw_config which are then used within the BL image

Usecase-3: Inform shared buffer for S/NS comms



This is an example how firmware modifies hw_config to pass info to other software.

1. BL2 authenticates and loads hw_config (linux kernel DTB)
2. BL2 authenticates and loads BL31, BL32 and BL33.
3. BL2 updates the right arguments to BL31, BL32 and BL33 to point hw_config
4. BL32 (TOS) modifies the hw_config, specifying the location/size of the shared buffer
5. Linux reads the hw_config containing the TOS shared buffer info

The same flow can be used to define a secure memory carveout for use by the secure world.

arm

What dynamic configuration is and is not

- The dynamic config files are part of the Chain-of-Trust and are authenticated as part of loading them to memory. They can be modified once loaded.
- Earlier BL images can modify the config files meant for later BL images.
  - The later BL images cannot change configuration of earlier BLs via this mechanism.
- Dynamic configuration (via config files) of each BL image is expected to be done only once during boot.
  - If runtime configuration EL3 firmware is required, then it must be done via an SMC.
  - The config memory can be reclaimed once the information is consumed.

arm

Enable PIE support in TF-A

Adding PIE support

- Position Independent executable (PIE) for BL images was one of the goals of dynamic configuration.
  - Once a BL image is PIE enabled, it can possibly be loaded to an address based on a dynamic configuration
  - Allows BL images to be moved around in memory without rebuilding the firmware. Reduces complex address calculations in the code.
  - Currently only FVP enables PIE in upstream.
- The relevant GCC support works from version 6 onwards
  - Add –fpie , - --no-dynamic-linker to the compiler and -pie to the linker.
  - Usually a loader is needed to fix the dynamic relocations and the GOT section
  - Each BL image does its own fixup in TF-A

Additional sections created for BL31 when PIE enabled for FVP (TF-Av2.1 release build)

```
Idx Name          Size
  3 .got          00000138
  4 .dynsym       000003d8
  5 .dynstr       00000375
  6 .hash         000001c0
  8 .dynamic      00000110
  9 .got.plt      00000018
 15 .rela.dyn     00000c00
```


arm

Finding the relevant sections for fixup

- The sections that need fixing up are the .got and addresses mentioned in .rela.dyn section
  - Needs to be done very early in the BL image
- The ELF information is not present in the firmware binaries. So we need other ways to locate these sections.
  - The location of .got and .rela.dyn sections can be extracted from .dynamic section.
  - There are symbols exporting the start of .got and .dynamic sections.
- Instead of parsing the .dynamic section, we added the start and end markers to the linker script for .got and .rela.dyn sections and refer them in the fixup code :

```
__GOT_START__ = .;
  .got . : {
  } >RAM
__GOT_END__ = .;
```

```
__RELA_START__ = .;
  .rela.dyn . : {
  } >RAM
__RELA_END__ = .;
```

Global Offset Table (GOT) fixup

- If a program requires direct access to the `absolute address` of a symbol, that symbol will have a global offset table entry.
  - Global variables, extern variables, string literals have GOT entries
- To fixup GOT entry, just add the difference between compiled address and runtime address to the entry.
- The assembler doesn’t generate GOT entries.
  - Assembly code is modified to use relative addressing instead of loading absolute address from literal pool ie :

```
-       ldr     x0, =__RW_START__
+       adrp    x0, __RW_START__
+       add     x0, x0, :lo12:__RW_START__
```

  - `adrp` solution shouldn’t be done for constant values (like BSS_SIZE, RW_SIZE etc) and is not dependant on the relative address. Because adrp is based on the current PC, doing adrp on SIZE macros adds the offset of (run addr – compile addr).

Dynamic relocations in .rela.dyn section

- *relocations* are entries in binaries that are left to be filled in later. A *relocation* in a binary is a descriptor which essentially says "determine the value of X, and put that value into the binary at offset Y" [1]
- Typical example of dynamic relocation entries are structures encapsulating callback functions. The function pointers are populated based on the compiled locations and relocation entries are created for them. This needs to be fixed up the loader at runtime.
- The dynamic relocations are populated in .rela.dyn section.
- There are different types of dynamic relocations.
  - The idea is to control the types of relocations in this section so that it becomes sensible enough for the simplistic loader to implement without the ELF information.
  - R_AARCH64_RELATIV type is enough for PIE in TF-A. Other types of relocations are caused by some code patterns and can be removed by modifying the code.

[1] https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html

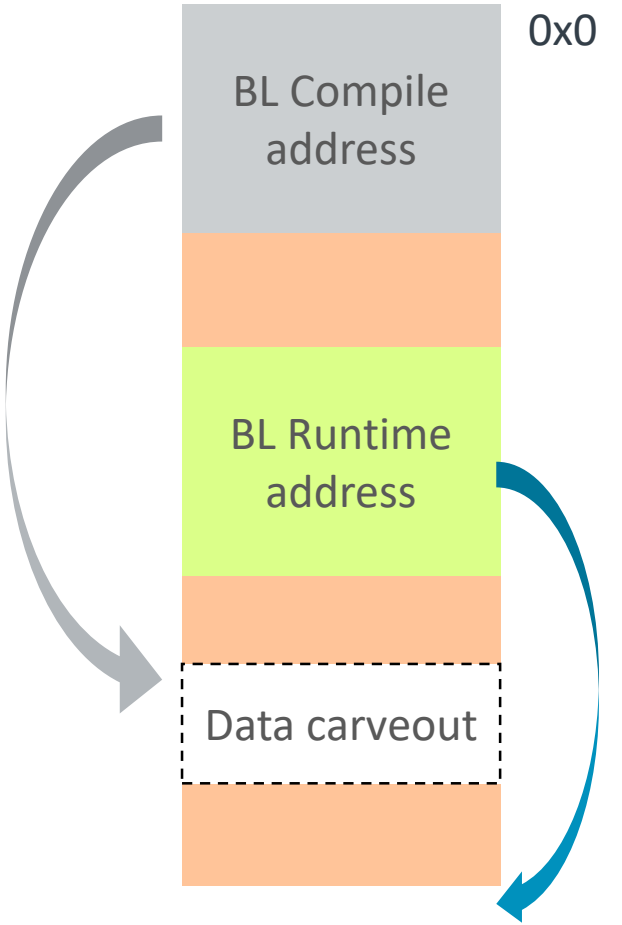
```
typedef struct
{
        Elf64_Addr r_offset; /* address of reference */
        Elf64_Xword r_info; /* symbol index and type of */
        Elf64_Sxword r_addend; /* constant part of expr */
} Elf64_Rela;
```

The RELA data structure.

arm

Some limitations of PIE support

- The current fixup code assumes that BL binary is a contiguous binary that is relocated as single blob at runtime.
- This is not the case for BL1.
- Referencing a section which is not relocated along with rest of binary
  - This happens when linker script is used to fix absolute addresses of certain sections disjoint from the rest of the binary.
  - Should not use relative addressing to access this section
- This cannot work with RECLAIM_INIT_CODE feature in TF-A
  - This is a runtime overlay wherein code used only during cold boot (.init) is reclaimed for data
  - FVP reclaims the .init section for secondary CPU stacks.



Referencing a section with fixed address

arm

Conclusion

- We think dynamic configuration can unlock more possibilities
  - CPU nodes, PSCI power state parameters can be populated by firmware
  - More opportunities to share memory between BL images and reduce peak memory footprint
- Need better adoption in partner platforms and TOS's
- Questions/Comments

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה

arm

Backup

Backup

- GOT examples

Consider this line of code :
NOTICE("BL31: %s\n", version_string);

It generates the following disassembly :
EL3:0x0000000004023474 : LDR x1,[x20,#0xb48]
EL3:0x0000000004023478 : MOV x0,x19
EL3:0x000000000402347C : BL tf_log ; 0x402678C

X20 points to GOT.

Another example : assert(base_addr >= PMF_TIMESTAMP_ARRAY_START);

Here the PMF_TIMESTAMP_ARRAY_START has a GOT entry

arm

Other relocation types in .rela.dyn section

- The last entry in the .rela.dyn section shows a different type of relocation.
  - But the destination is within .text section.
- This is created by code here which is solved by changing the ldr to adrp/adr instructions.
- Weak references creates the same problem (here).
  - Removing the weak reference solved the problem

```
000004013bb0  000000000403 R_AARCH64_RELATIV                    4018f58
000004013bb8  000000000403 R_AARCH64_RELATIV                    4018f50
000004013bc0  000000000403 R_AARCH64_RELATIV                    4018fc0
000004013bc8  000000000403 R_AARCH64_RELATIV                    40129f0
000004013bd0  000000000403 R_AARCH64_RELATIV                    401f040
000004013bd8  000000000403 R_AARCH64_RELATIV                    40129e8
000004013be0  000000000403 R_AARCH64_RELATIV                    4018f40
000004013be8  000000000403 R_AARCH64_RELATIV                    40129b8
000004013bf0  000000000403 R_AARCH64_RELATIV                    40129f0
000004013bf8  000000000403 R_AARCH64_RELATIV                    40125f8
000004013c00  000000000403 R_AARCH64_RELATIV                    40129b8
000004013c08  000000000403 R_AARCH64_RELATIV                    40129d0
000004013c10  000000000403 R_AARCH64_RELATIV                    4018a08
000004013c18  000000000403 R_AARCH64_RELATIV                    40129b8
000004013c20  000000000403 R_AARCH64_RELATIV                    4018f48
00000400e708  000700000101 R_AARCH64_ABS64   0000000004018f60 mmu_cfg_params + 0
sobmat01@e110881-lin:/work/github/arm-trusted-firmware$
```

`readelf –rd` dump showing the other relocation in .rela.dyn section

arm

Fixing R_AARCH64_RELATIV relocation

- According to ELF-64 specification, the RELA data structure in .rela.dyn section is as follows:
  - typedef struct
    {
    Elf64_Addr r_offset; /* address of reference */
    Elf64_Xword r_info; /* symbol index and type of relocation (in this case 0x403 which corresponds to R_AARCH64_RELATIV). */
    Elf64_Sxword r_addend; /* constant part of expression */
    } Elf64_Rela;
    sizeof(Elf64_Rela) is 24 bytes.
- The fixup operation for each entry in this section is as follows:
  - if (r_info == R_AARCH64_RELATIV)
    *(r_offset + diff(S)) = r_addend + diff(S);
    where diff(S) is the difference between the compiled address and runtime address.
  - The code can be seen here
- 2 new symbols are introduced into the linker script to detect the start and end of .rela.dyn section at runtime : RELA_START and RELA_END