10 MIN

DATE

8 MARCH 2022

FIRMWARE ARM BINARY QILING

AUTHOR

ZI0BLACK

I'm zi0black, Security Researcher and Penetration Tester at Shielder. I love to turn IoT devices in expensive paperweights.

AUTHOR

THEZERO

Security Researcher and Senior Penetration Tester at Shielder.
In the office I'm the one with the soldering iron.

Reversing embedded device bootloader (U-Boot) - p.1

This blog post is not intended to be a "101" ARM firmware reverse-engineering tutorial or a guide to attacking a specific IoT device. The goal is to share our experience and, why not, perhaps save you some precious hours and headaches.

"Bootrom"

In this two posts series, we will share an analysis of some aspects of reversing a low-level binary. Why? Well, we have to admit we struggled a bit to collect the information to build the basic knowledge about this topic and the material we found was often not comprehensive enough, or many aspects were taken for granted. For this reason, we share here what we learned from multiple sources and try to collect them in these posts, while also trying to give some context and analyze the more complex or cryptic aspects.

Some context before the 🛫:

- The CPU is an ARM-Cortex A7.
- The bootloader is a customized U-Boot (the binary is stripped).
- We found the datasheet of the SOC.
- We got the firmware image from the vendor site.
- Kernel and rootfs are encrypted, presumably by a “custom” cryptographic approach
- The IoT device doesn’t use ARM Trust Zone.
- The device has secure boot enforced.

The main goal was to reverse the custom crypto function, retrieve the encryption key, and decrypt the kernel image. The adventured ended slightly in a different way, but, spoiler alert, we did manage to decrypt the kernel image.

What is a bootloader?

The first program that runs when a computer starts up is the bootloader, which loads the operating system. It is typically stored on an EEPROM or a NOR flash memory (a type of persistent flash memory) part of the computer hardware. Its function is to initialize the various system components: from the CPU registers to the device controllers and the central memory content. The start-up program needs to locate, load into main memory and then transfer control to the operating system, so that it can then start offering services to the system.

Some computer systems use a multistage boot process: when the computer is first turned on, a small bootloader located in non-volatile memory, known as the *BIOS*, is executed; this initial program then loads a second bootloader situated in a fixed area of the disk (called the boot block). The second start-up program is more complex (think something like *Grub*, which is many thousands of lines of code) than its loader and does all the required heavy lifting of setting up enough support to more easily load the operating system.

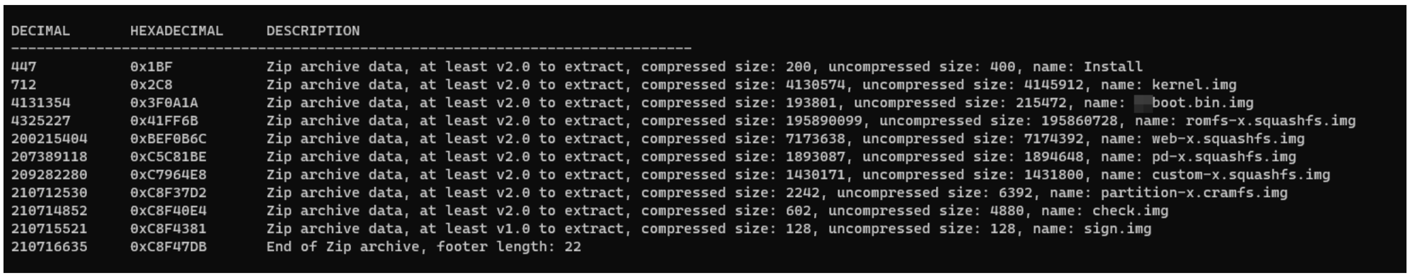
For a real-world example, we suggest looking into this external resource: ARM boot process.

We won’t go into the technical details of the Das U-Boot implementation, but it’s enough to say that U-Boot is an open-source primary bootloader used mainly in embedded devices.

NOTE: “Primary bootloader” doesn’t mean that U-Boot must be the first-stage bootloader - it could be used at any stage

L◔_◔king around

After downloading the firmware, we used Binwalk to extract it.

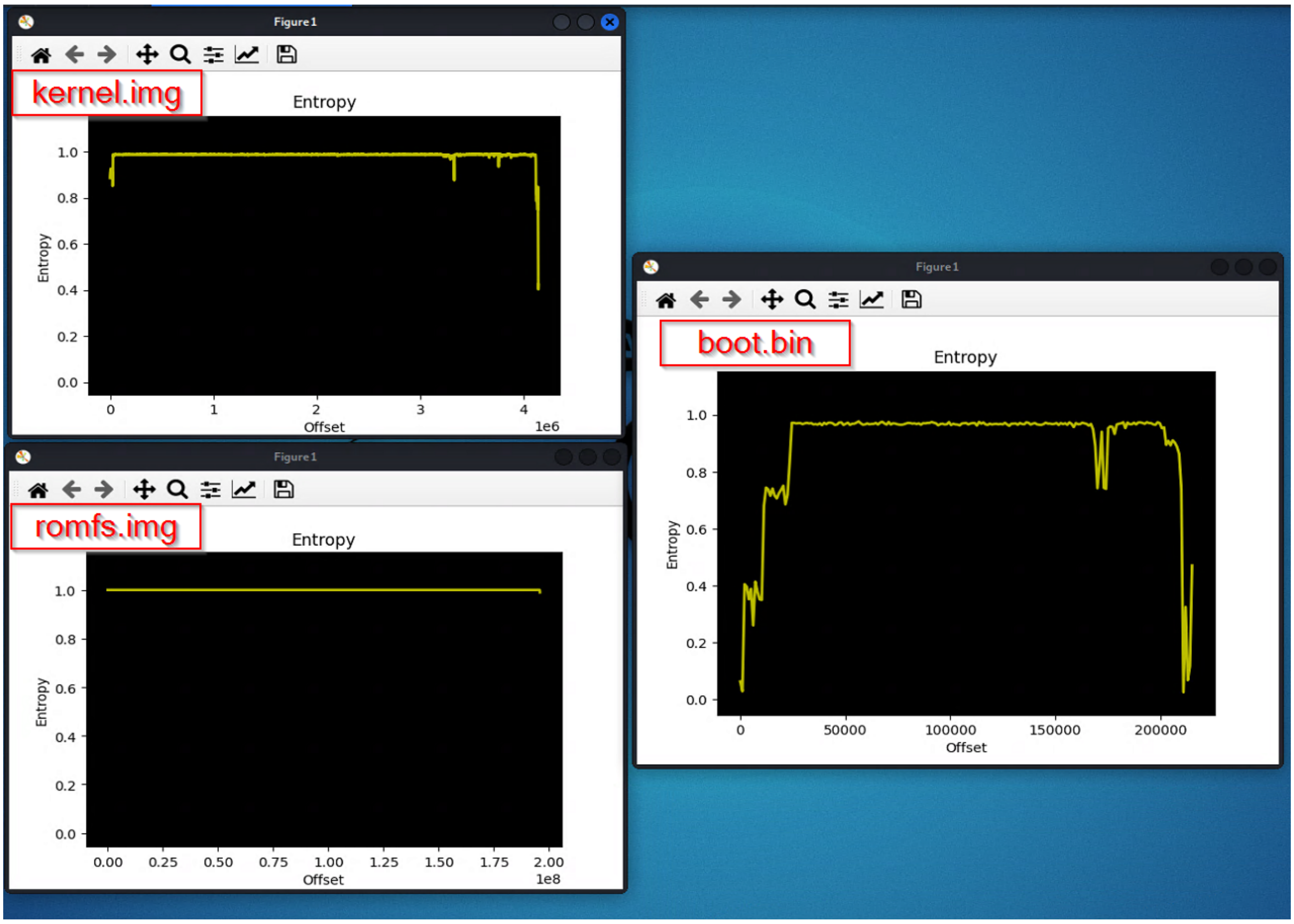


```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
447           0x1BF           Zip archive data, at least v2.0 to extract, compressed size: 200, uncompressed size: 400, name: Install
712           0x2C8           Zip archive data, at least v2.0 to extract, compressed size: 4130574, uncompressed size: 4145912, name: kernel.img
4131354       0x3F0A1A        Zip archive data, at least v2.0 to extract, compressed size: 193801, uncompressed size: 215472, name: [redacted]boot.bin.img
4325227       0x41FF6B        Zip archive data, at least v2.0 to extract, compressed size: 195890099, uncompressed size: 195860728, name: romfs-x.squashfs.img
200215404     0xBEF0B6C       Zip archive data, at least v2.0 to extract, compressed size: 7173638, uncompressed size: 7174392, name: web-x.squashfs.img
207389118     0xC5C81BE       Zip archive data, at least v2.0 to extract, compressed size: 1893087, uncompressed size: 1894648, name: pd-x.squashfs.img
209282280     0xC7964E8       Zip archive data, at least v2.0 to extract, compressed size: 1430171, uncompressed size: 1431800, name: custom-x.squashfs.img
210712530     0xC8F37D2       Zip archive data, at least v2.0 to extract, compressed size: 2242, uncompressed size: 6392, name: partition-x.cramfs.img
210714852     0xC8F40E4       Zip archive data, at least v2.0 to extract, compressed size: 602, uncompressed size: 4880, name: check.img
210715521     0xC8F4381       Zip archive data, at least v1.0 to extract, compressed size: 128, uncompressed size: 128, name: sign.img
210716635     0xC8F47DB       End of Zip archive, footer length: 22
```

Unfortunately that didn't produce the expected results, as it didn't recognize nor extract the (expected) various partitions.

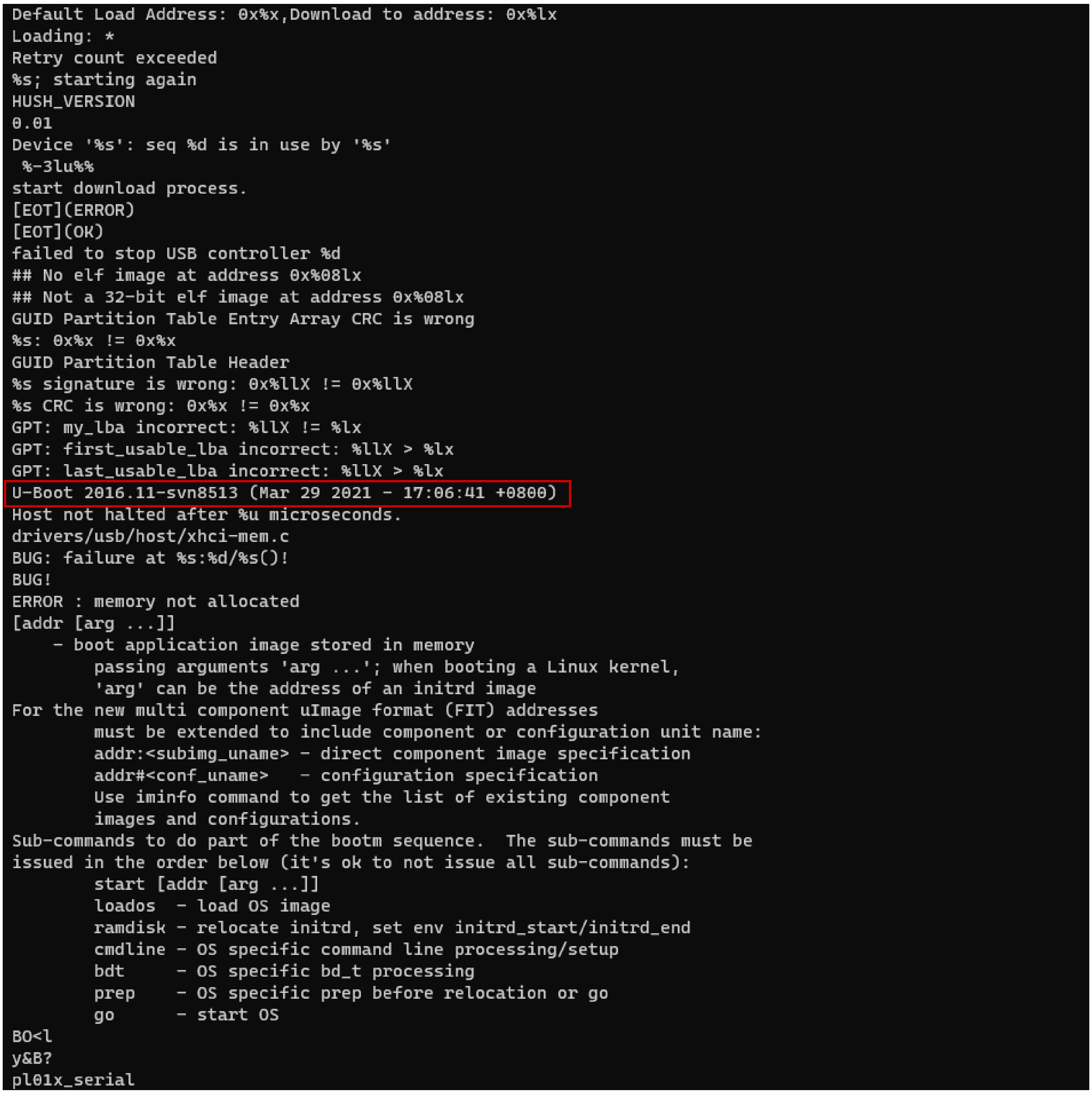
What this usually means is that files are probably encrypted somehow or have - unlikely, but possible - a custom format. We can verify the first assumption by checking the entropy of the single file. Binary files tend to have frequent repetitions of certain instructions (e.g. prologues, nop sequences, etc) and data structures are **hardly random**. Long sequences of zeroes are also quite common in the data segment, when not everything can be deferred to bss. On the contrary, an encrypted file will have nearly perfect entropy, since that's kind of the goal of a robust encryption scheme 😉

To make this check, one can use the Binwalk `--entropy` flag, to check the entropy of all the firmware files. As you can notice from the graph in the picture, most of the files have an almost perfectly flattered Y-axis entropy value of one (1) - this confirms they are encrypted.

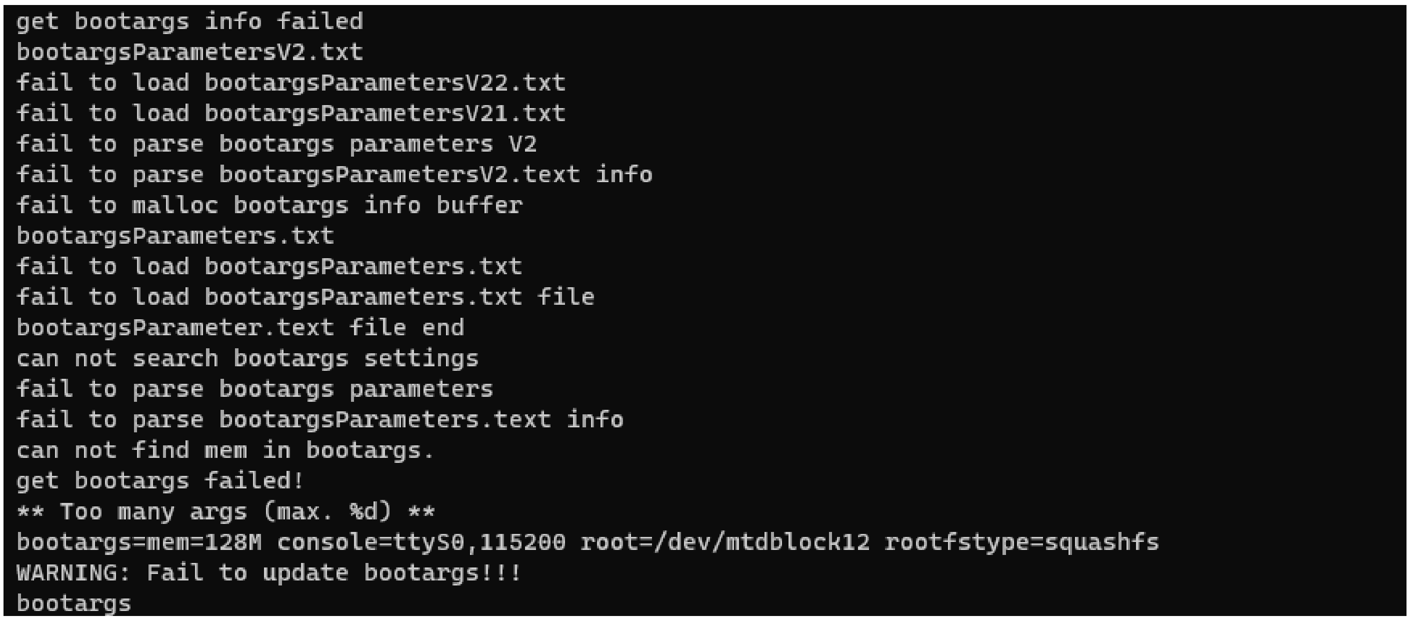


The `boot.bin` file instead stands out as it doesn't have a high constant entropy. See those drops? That's the repetitions we talked about. We can therefore make an educated guess about the fact that the clear-text bootloader is the one in charge of decrypting the other partitions during the boot process, right before starting kernel execution. We still didn't know how much of the decryption logic is in the bootloader: it could be decrypting everything or just the kernel, who would then proceed to use its own set of keys/algorithms to decrypt the rest of filesystem. This latter approach is not uncommon both in encrypted and verified chains.
Armed with this current assumption, we extracted the boot image from the firmware with Binwalk (no black magic, only `binwalk -e firmware.bin`) and then run Binwalk again to extract the `boot.bin` file.

Let's run `strings boot.bin` just to make sure we're not completely bonkers.


```
Default Load Address: 0x%x,Download to address: 0x%lx
Loading: *
Retry count exceeded
%s; starting again
HUSH_VERSION
0.01
Device '%s': seq %d is in use by '%s'
 %-3lu%%
start download process.
[EOT](ERROR)
[EOT](OK)
failed to stop USB controller %d
## No elf image at address 0x%08lx
## Not a 32-bit elf image at address 0x%08lx
GUID Partition Table Entry Array CRC is wrong
%s: 0x%x != 0x%x
GUID Partition Table Header
%s signature is wrong: 0x%llX != 0x%llX
%s CRC is wrong: 0x%x != 0x%x
GPT: my_lba incorrect: %llX != %lx
GPT: first_usable_lba incorrect: %llX > %lx
GPT: last_usable_lba incorrect: %llX > %lx
U-Boot 2016.11-svn8513 (Mar 29 2021 - 17:06:41 +0800)
Host not halted after %u microseconds.
drivers/usb/host/xhci-mem.c
BUG: failure at %s:%d/%s()!
BUG!
ERROR : memory not allocated
[addr [arg ...]]
    - boot application image stored in memory
        passing arguments 'arg ...'; when booting a Linux kernel,
        'arg' can be the address of an initrd image
For the new multi component uImage format (FIT) addresses
        must be extended to include component or configuration unit name:
        addr:<subimg_uname> - direct component image specification
        addr#<conf_uname>   - configuration specification
        Use iminfo command to get the list of existing component
        images and configurations.
Sub-commands to do part of the bootm sequence.  The sub-commands must be
issued in the order below (it's ok to not issue all sub-commands):
        start [addr [arg ...]]
        loados  - load OS image
        ramdisk - relocate initrd, set env initrd_start/initrd_end
        cmdline - OS specific command line processing/setup
        bdt     - OS specific bd_t processing
        prep    - OS specific prep before relocation or go
        go      - start OS
BO<l
y&B?
pl01x_serial
```

Yay - we have a proper U-Boot binary! Let's check the boot arguments: `strings boot.bin | grep args`.

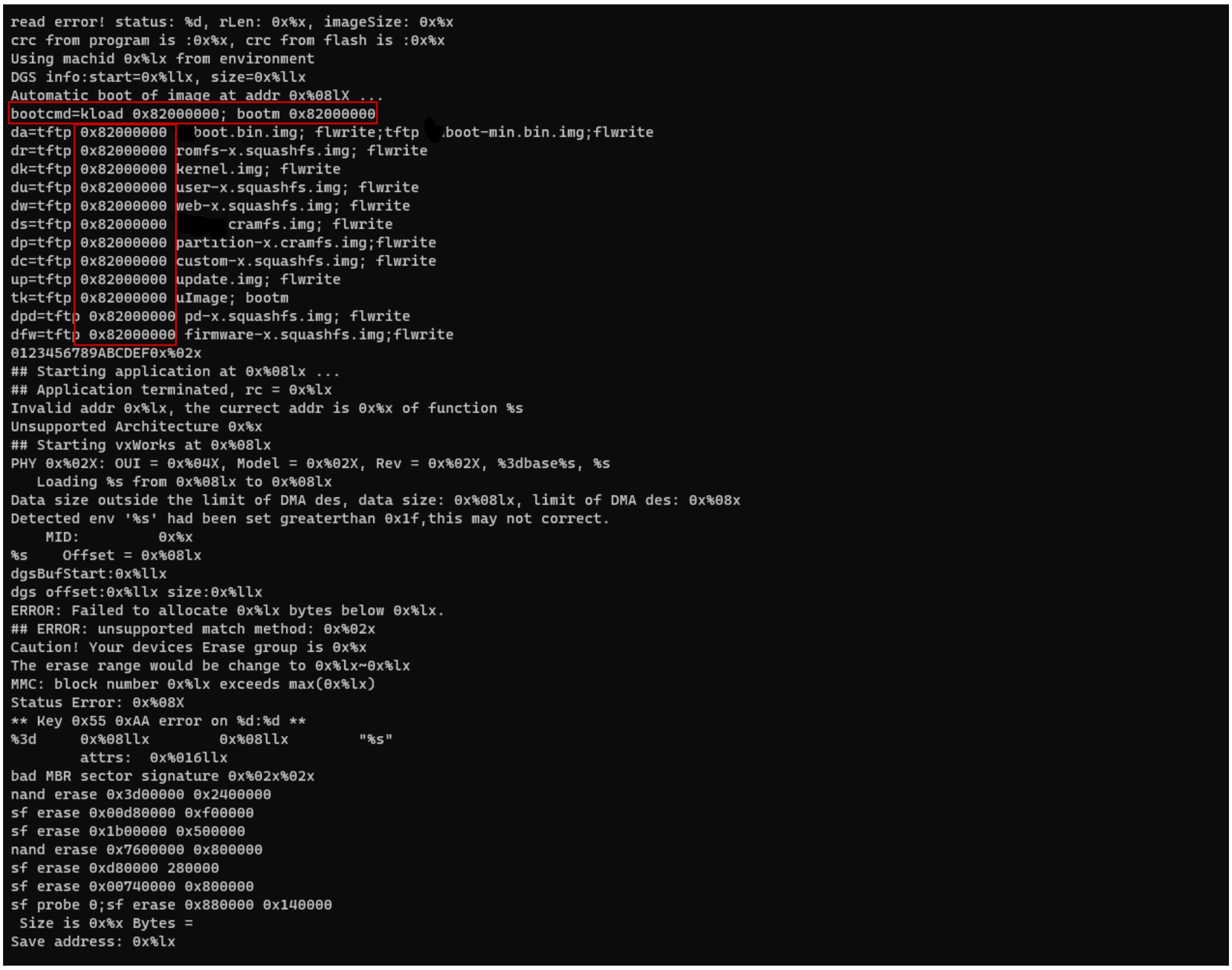


```
get bootargs info failed
bootargsParametersV2.txt
fail to load bootargsParametersV22.txt
fail to load bootargsParametersV21.txt
fail to parse bootargs parameters V2
fail to parse bootargsParametersV2.text info
fail to malloc bootargs info buffer
bootargsParameters.txt
fail to load bootargsParameters.txt
fail to load bootargsParameters.txt file
bootargsParameter.text file end
can not search bootargs settings
fail to parse bootargs parameters
fail to parse bootargsParameters.text info
can not find mem in bootargs.
get bootargs failed!
** Too many args (max. %d) **
bootargs=mem=128M console=ttyS0,115200 root=/dev/mtdblock12 rootfstype=squashfs
WARNING: Fail to update bootargs!!!
bootargs
```

Nothing special, but now we know where the squashfs rootfs is stored: `/dev/mtdblock12`.

What else can we learn? Well, let's look for some addresses to get an idea of how this thing will look like in memory. Looking at the datasheet we found a table with all the device addresses mapped and - not that surprisingly if you are used to ARM - that the RAM starts at `0x8000_0000`.

Time for some extra `grep`-fu: `strings u-boot.bin | grep 0x`



```
read error! status: %d, rLen: 0x%x, imageSize: 0x%x
crc from program is :0x%x, crc from flash is :0x%x
Using machid 0x%lx from environment
DGS info:start=0x%llx, size=0x%llx
Automatic boot of image at addr 0x%08lX ...
bootcmd=kload 0x82000000; bootm 0x82000000
da=tftp 0x82000000 [redacted]boot.bin.img; flwrite;tftp [redacted]boot-min.bin.img;flwrite
dr=tftp 0x82000000 romfs-x.squashfs.img; flwrite
dk=tftp 0x82000000 kernel.img; flwrite
du=tftp 0x82000000 user-x.squashfs.img; flwrite
dw=tftp 0x82000000 web-x.squashfs.img; flwrite
ds=tftp 0x82000000 [redacted]cramfs.img; flwrite
dp=tftp 0x82000000 partition-x.cramfs.img;flwrite
dc=tftp 0x82000000 custom-x.squashfs.img; flwrite
up=tftp 0x82000000 update.img; flwrite
tk=tftp 0x82000000 uImage; bootm
dpd=tftp 0x82000000 pd-x.squashfs.img; flwrite
dfw=tftp 0x82000000 firmware-x.squashfs.img;flwrite
0123456789ABCDEF0x%02x
## Starting application at 0x%08lx ...
## Application terminated, rc = 0x%lx
Invalid addr 0x%lx, the currect addr is 0x%x of function %s
Unsupported Architecture 0x%x
## Starting vxWorks at 0x%08lx
PHY 0x%02X: OUI = 0x%04X, Model = 0x%02X, Rev = 0x%02X, %3dbase%s, %s
   Loading %s from 0x%08lx to 0x%08lx
Data size outside the limit of DMA des, data size: 0x%08lx, limit of DMA des: 0x%08x
Detected env '%s' had been set greaterthan 0x1f,this may not correct.
    MID:         0x%x
%s    Offset = 0x%08lx
dgsBufStart:0x%llx
dgs offset:0x%llx size:0x%llx
ERROR: Failed to allocate 0x%lx bytes below 0x%lx.
## ERROR: unsupported match method: 0x%02x
Caution! Your devices Erase group is 0x%x
The erase range would be change to 0x%lx~0x%lx
MMC: block number 0x%lx exceeds max(0x%lx)
Status Error: 0x%08X
** Key 0x55 0xAA error on %d:%d **
%3d     0x%08llx        0x%08llx        "%s"
        attrs:  0x%016llx
bad MBR sector signature 0x%02x%02x
nand erase 0x3d00000 0x2400000
sf erase 0x00d80000 0xf00000
sf erase 0x1b00000 0x500000
nand erase 0x7600000 0x800000
sf erase 0xd80000 280000
sf erase 0x00740000 0x800000
sf probe 0;sf erase 0x880000 0x140000
 Size is 0x%x Bytes =
Save address: 0x%lx
```

Okay, so if the RAM starts at `0x8000_0000` it makes sense that the kernel is loaded at `0x8200_0000`. But what is loaded from `0x8000_0000` to `0x8200_0000`? And where is U-Boot loaded? It should be loaded in a fixed address since it's the bootloader!

zi0black: *I've just taken my operating systems exams at university, and I'm very confident about this. I could have probably looked more deeply into the datasheet or other documentation to find out, but I chose a different way.*

Dissection of a bare metal binary (ARM)

Binary files are the essence of what is loaded and interpreted/executed by a computer. In their essence, they are just a sequence of bytes that gets splatted in memory and has enough information to get the execution going. In practice, there can be much more to them in order to support *dynamic linking*, *shared libraries*, *runtime relocations* and all the other flexibility we almost take for granted when we compile a binary on one system and run it on another, or see it pick up a fixed library after a security update. On top of that, a program needs to do something useful to fulfill its existential meaning (yeah, let's get philosophical). To achieve this it will most likely need to interact with the system, allocate some memory, maybe store some data to disk. We **don't** expect all binaries to implement this logic: the operating system is there for them.

The picture we just described above is the one you normally encounter with an (ELF) running on a Linux environment. Of course, bringing up all this ecosystem has a non trivial cost: you need a fully working operating system, a dynamic linker and all the libraries. In **IoT** or other memory constrained environments - or in cases where you don't want all these layers of abstractions in the way (think some specialized cloud workload or similar) - one can have a single binary do everything it needs and just what it needs. This is the central idea behind **Bare Machine Computing** (BMC). In the BMC paradigm, applications run **without** the support of any **operating system** (OS) or centralized Kernel, i.e., no intermediary software is loaded on the bare machine prior to running applications.

All we get with *BMC* is a **big static flat file** that will just start executing and manage memory, handle interrupts and (if needed) access hardware directly by itself. It's quite common for these binaries, since they are the only entity in execution, to not have to implement any form of virtual memory, as there's really no "separation" that needs to be created, nor there's the need to go beyond the amount of installed memory with some form of paging. For our analysis, this means that if we are dealing with a bare-metal binary, we will find lots of information about the memory layout directly where we would "normally" (e.g. with an *ELF* file) find runtime-resolved relocations. Goes without saying, U-Boot is a bare-metal binary.

Interrupts

Let's briefly touch on interrupts, too, before moving to our target binary and understand its structure.

Hardware components can generate an interruption at **any time** by sending a signal to the CPU, usually via the system bus (there can be many buses within a processing system, but the system bus is the primary communication path between the core components). Interrupts are also used for many other purposes and are crucial for the interactions between the operating systems and the underlying hardware. When the CPU receives an interrupt signal, it stops the current processing and immediately jumps to some fixed memory region.

> It should be noted that there is no black magic behind the change of context due to the execution of an interrupt handler, except for CPU peculiarities if present (e.g. an additional set of registers that saves to the programmer some of the context switch heavy lifting). The interrupt handler is responsible to save the current state/registers (context) and later restore them to correctly resume execution of the interrupted instruction stream once the servicing is done.

This "memory region" is basically a table of fixed size entries that contain either the address of or directly the first instructions of the dedicated interrupt service routine. Depending on the size of each entry and the format, some amount of instructions could be stashed directly there. These may or may not be sufficient to completely handle the interrupt: generally they are not and the very first thing that is done is to branch somewhere else to start handling the interrupt. In case of a stored address, the CPU just directly loads it into the program counter.

Interrupts Vector Table

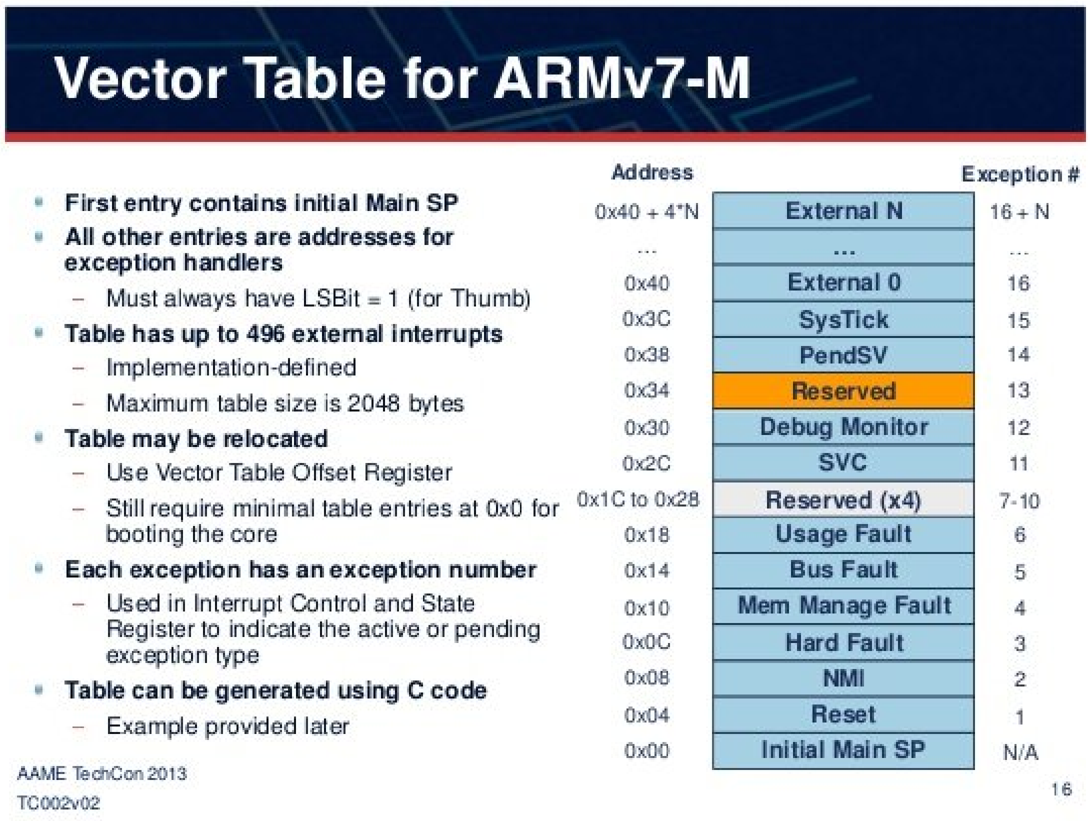
> UPDATE - 5th of April 2022: I should thank @Rekreker, that pointed out the improper usage of the term Interrupts Vector Table (IVT) in the context of an ARMv7-A/R CPU.🤝
>
> Let's add some context before we start using the term IVT: "ARMv7-A uses the generic term exception to refer, in general terms, to interrupts and some other exception types like CPU errors. An interrupt is called an IRQ exception in ARMv7-A, so that's the term the manual names a lot. When an ARMv7-A CPU takes an exception, it transfers control to an instruction located at the appropriate location in the vector table, depending on the exception type. The very first code we wrote for startup began with the vector table."
>
> – umanovskis - Bare-metal C programming on ARM

This pointer table, also known as the Interrupt Vector Table (IVT), is generally stored in the lowest part of the central memory (e.g., the first n locations, we will see it later). The table entries have an index, which is the same included in the interrupts, allowing a fast lookup.

Interrupts are very similar to system exceptions, having the main difference in which is the component generating them: the first are generated by architecture-specific peripheral modules, while the second by the CPU. They are also unpredictable, whereby exceptions are deterministic and in response to certain program behavior.

When it comes to an OS the kernel handles interrupts but in bare-metal binaries, such as U-Boot, the single binary **should contain and handle the IVT**. The IVT is therefore a great starting point while analyzing a raw binary. In our specific case, an ARM device, we know that the IVT should be placed (it could be relocated) at the beginning of the address space: 0x00, 0x04, 0x08, … This means that finding the IVT would bring us to the beginning of the binary!

An graphical representation of an Interrupt Vector Table follows:



Vector Table for ARMv7-M

- First entry contains initial Main SP
- All other entries are addresses for exception handlers
  - Must always have LSBit = 1 (for Thumb)
- Table has up to 496 external interrupts
  - Implementation-defined
  - Maximum table size is 2048 bytes
- Table may be relocated
  - Use Vector Table Offset Register
  - Still require minimal table entries at 0x0 for booting the core
- Each exception has an exception number
  - Used in Interrupt Control and State Register to indicate the active or pending exception type
- Table can be generated using C code
  - Example provided later

Address		Exception #
0x40 + 4*N	External N	16 + N
...	...	...
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

AAME TechCon 2013
TC002v02

16

After searching for a while on GitHub, we found some code that confirmed the structure of the interrupt vector table for the specific SoC/Board we were analyzing. It slightly differs from the previous one as it follows some ARM SoC specific characteristics.

```
.globl _start
_start: b       reset
ldr     pc, _undefined_instruction
ldr     pc, _software_interrupt
ldr     pc, _prefetch_abort
ldr     pc, _data_abort
ldr     pc, _not_used
ldr     pc, _irq
ldr     pc, _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt:    .word software_interrupt
_prefetch_abort:        .word prefetch_abort
_data_abort:            .word data_abort
_not_used:              .word not_used
_irq:                   .word irq
_fiq:                   .word fiq
_pad:                   .word 0x12345678 /* now 16*4=64 */
```

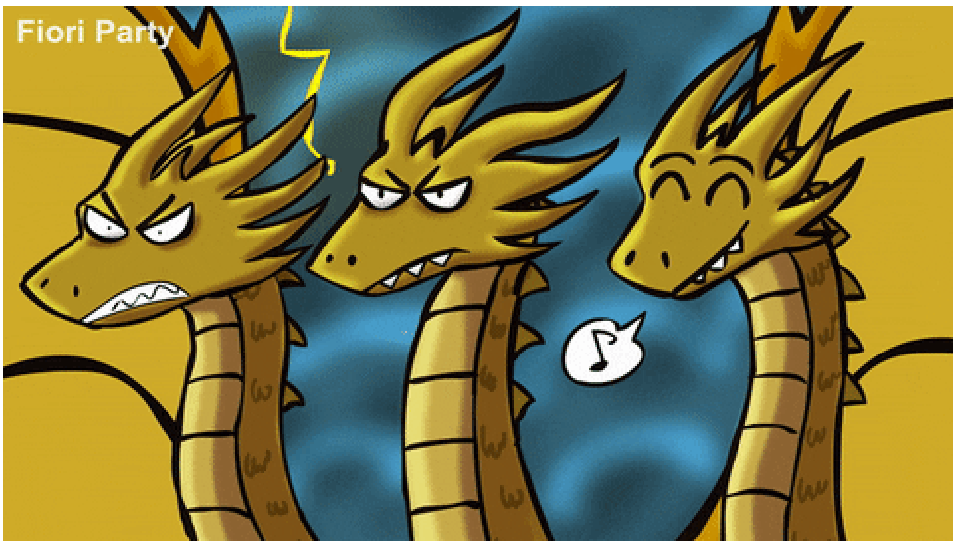
> When the processor is reset then hardware sets the pc to 0x0000 and starts executing by fetching the instruction at 0x0000. When an undefined instruction is executed or tries to be

> executed the hardware responds by setting the pc to 0x0004 and starts executing the instruction at 0x0004. irq interrupt, the hardware finishes the instruction it is executing starts executing the instruction at address 0x0018.
>
> – **old_timer** on stackoverflow

Wanna dig deeper into IVTs, Interrupts and Exceptions handling, and how an ARM CPU boots, check the Resources section!

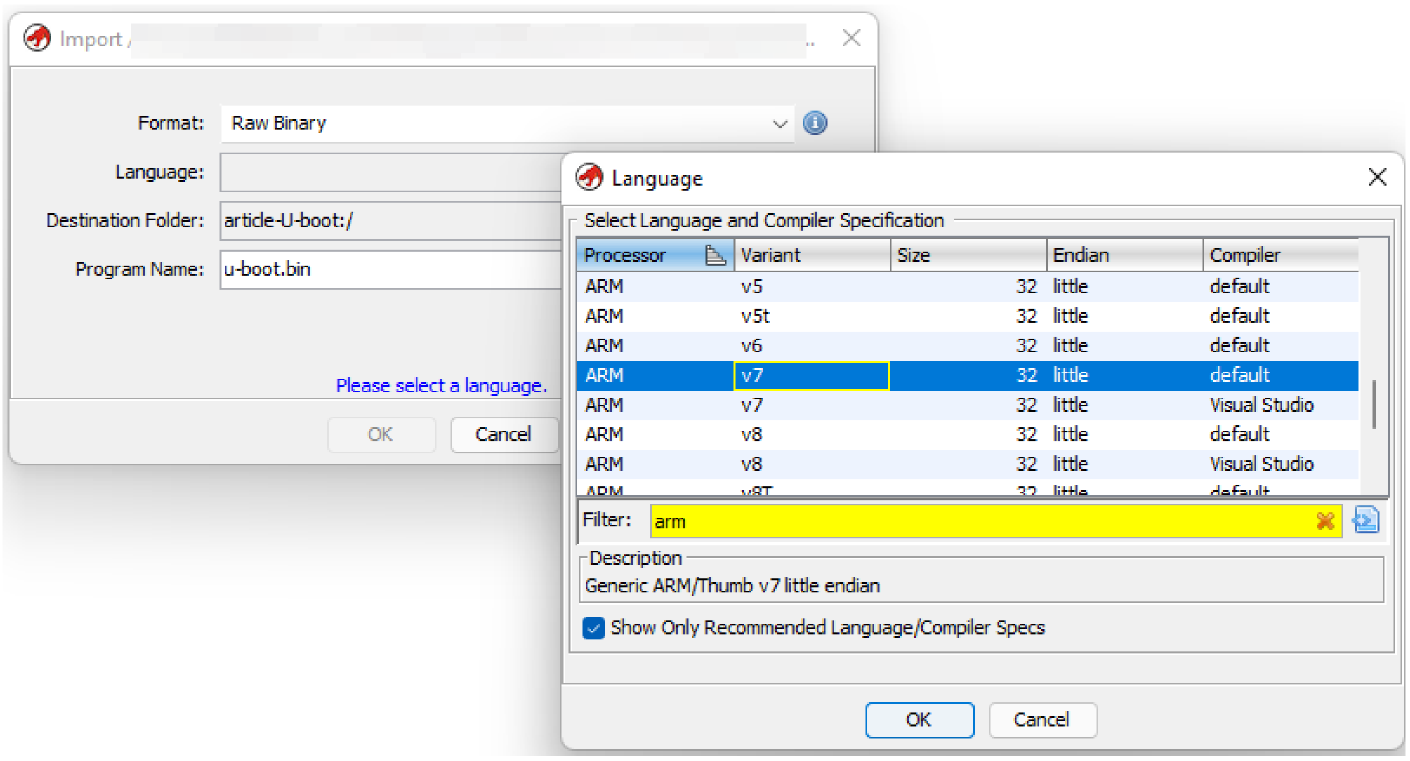
Evocation of the three-head dragon

Ghidra configuration - round 1°



It is time to start reversing our U-Boot binary in Ghidra.
Ghidra, when we import U-Boot, will load it as a raw binary, and we should instruct it about how it about the CPU architecture. Luckily for us, we knew from the datasheet that the SoC is composed by two ARM-A CPUs. A quick Google search of ARM-A17 reveals that it is ARMv7-EL based.

We can configure the loader accordingly with the information we found.

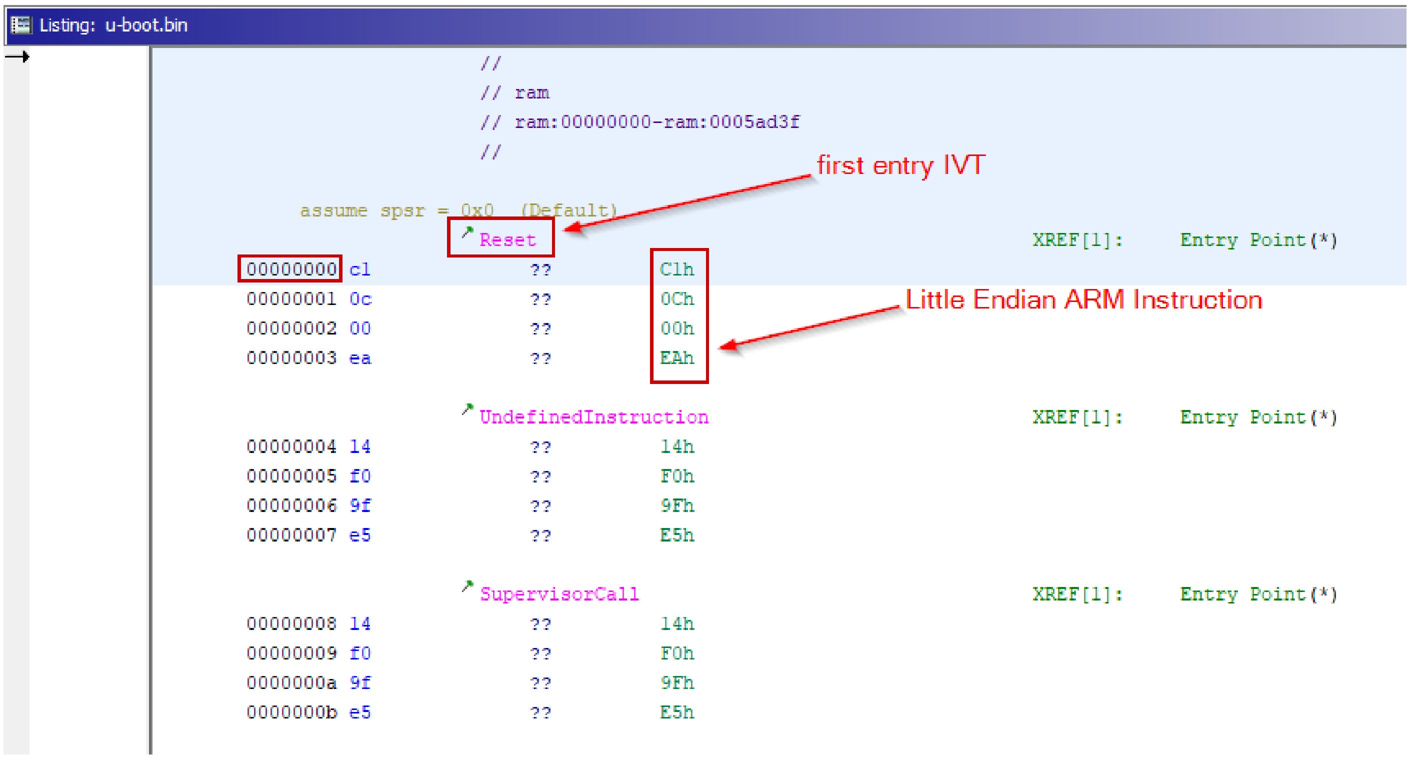

Import
Format: Raw Binary
Language:
Destination Folder: article-U-boot:/
Program Name: u-boot.bin
Please select a language.
OK
Cancel
Language
Select Language and Compiler Specification

Processor	Variant	Size	Endian	Compiler
ARM	v5	32	little	default
ARM	v5t	32	little	default
ARM	v6	32	little	default
ARM	v7	32	little	default
ARM	v7	32	little	Visual Studio
ARM	v8	32	little	default
ARM	v8	32	little	Visual Studio

Filter: arm
Description
Generic ARM/Thumb v7 little endian
Show Only Recommended Language/Compiler Specs
OK
Cancel

Manual analysis

Ghidra is smart enough to recognize the IVT even before the in-depth analysis.



```
                         //
                         // ram
                         // ram:00000000-ram:0005ad3f
                         //

           assume spsr = 0x0  (Default)
                       Reset                          XREF[1]:     Entry Point(*)
00000000 c1            ??        C1h
00000001 0c            ??        0Ch
00000002 00            ??        00h
00000003 ea            ??        EAh

                       UndefinedInstruction           XREF[1]:     Entry Point(*)
00000004 14            ??        14h
00000005 f0            ??        F0h
00000006 9f            ??        9Fh
00000007 e5            ??        E5h

                       SupervisorCall                 XREF[1]:     Entry Point(*)
00000008 14            ??        14h
00000009 f0            ??        F0h
0000000a 9f            ??        9Fh
0000000b e5            ??        E5h
```

The highlighted *DWORD*-sized hexadecimal sequence is an unconditional ARM instruction.

Statistically, unconditional instructions are the most common ones, and they can be recognized by the first bit, which is in the `0xE0`-`0xEF` range (remember that the architecture is little-endian, therefore the most significant bit is the last one).

A small digression on the size of the instructions.

> The Arm architecture supports three instruction sets: A64, A32 and T32.
>
> The A64 and A32 instruction sets have fixed instruction lengths of 32-bits. The T32 instruction set was introduced as a supplementary set of 16-bit instructions that supported improved code density for user code. Over time, T32 evolved into a 16-bit and 32-bit mixed-length instruction set. As a result, the compiler can balance performance and code size trade-off in a single instruction set. ARM Developer

The difference between two equivalent instructions is how they are fetched and interpreted prior to execution, not how they function. Since the expansion from 16-bit to 32-bit instruction is accomplished via dedicated hardware within the chip, it does not slow execution. However, the narrower 16-bit instructions offer memory advantages in terms of occupied space. Now let's say that in our case the CPU is using ARM *"ARM"* (and not ARM *Thumb*) instructions, so we are working with instructions of **32bit** in size. Remember that the CPU can switch to and from Thumb mode at runtime.

Let's convert the instruction into its binary representation:

```
ea -> 11101010
00 -> 00000000
0c -> 00001100
c1 -> 11000001
```

The Branch ARM instruction is structured as follows:

- bits from 31-28: condition
- bits from 27-25: fixed sequence
- bit 24: link bit
- bits from 23 to 0: offset represented in Two's complement

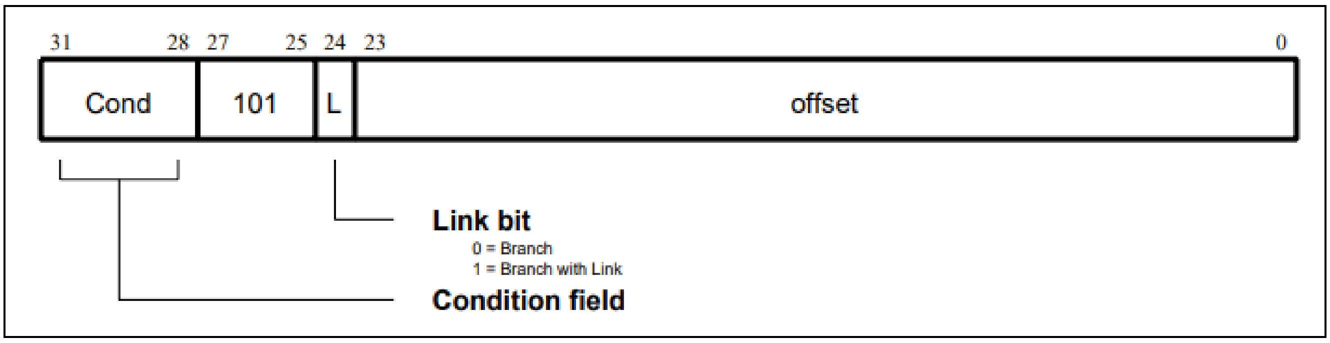


Figure 4-3: Branch instructions

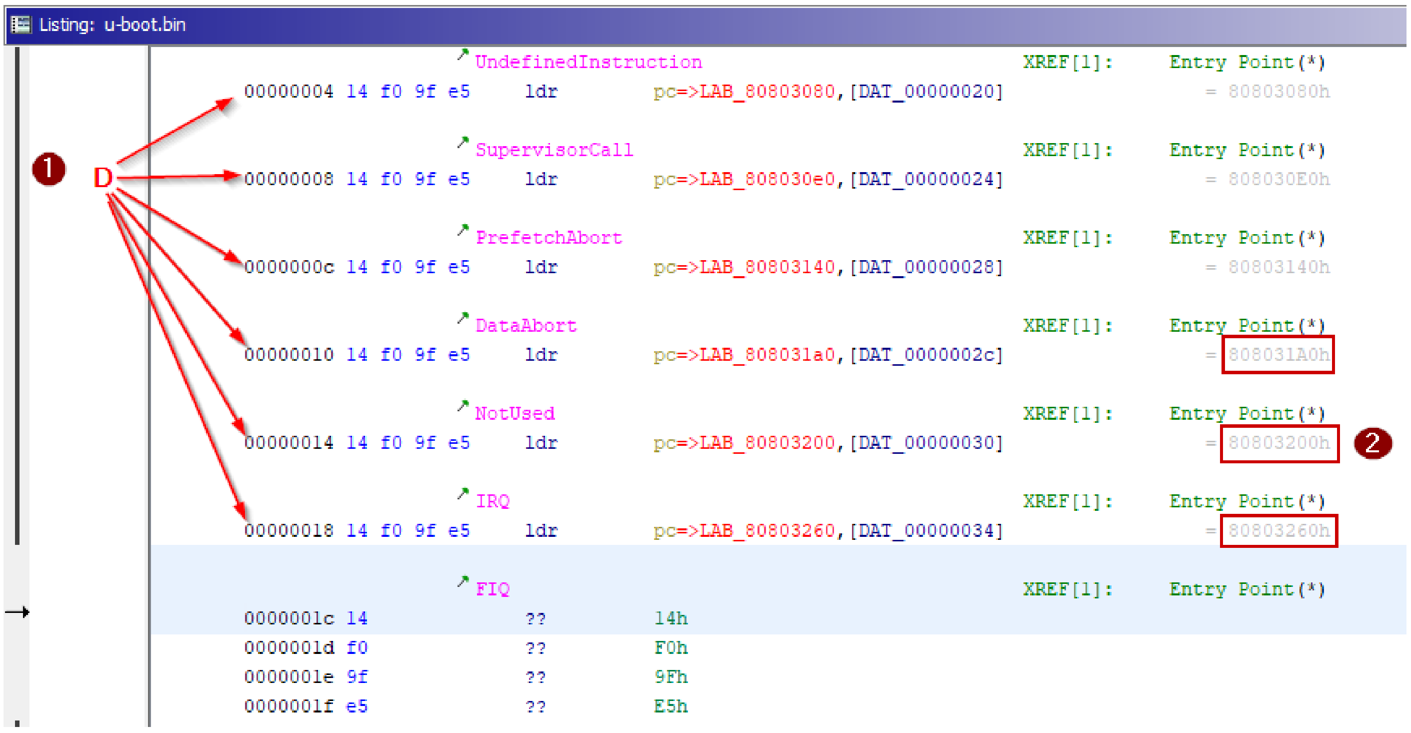
This ARM instruction is a branch (represented with a `B` in the ASM code), and its function is to jump (if a condition is met) to an address (PC + offset), changing the execution flow.
The first 4 bits are `1110` which corresponds to "ignore all CPU flags": aka unconditional branch.
The next 3 bits are fixed to `101` in branch instructions.
The 7th bit indicates if the branch should store a link to return or not. If it is set to `1` then an address

is stored in the R14 register and the CPU jumps back to that address when the function execution is completed. In our case the bit is set to `0` so it will just branch without storing anything to the R14 register.
The last bits store a Two's complement 24-bit offset. This is shifted left by 2 bits for memory alignment purposes, sign-extended to 32 bits, and added to the PC+8 to obtain the memory address to jump to.

Using Ghidra's decompiler we can decompile the first DWORD and observe that each instruction corresponds to a differential interrupt.
Despite the first interrupt ("reset"), all the others do an offset load inside the PC through the `LDR` assembly instruction and this is particularly interesting for us since we know where the RAM start (`0x8000_0000`) but not the offset of our binary.

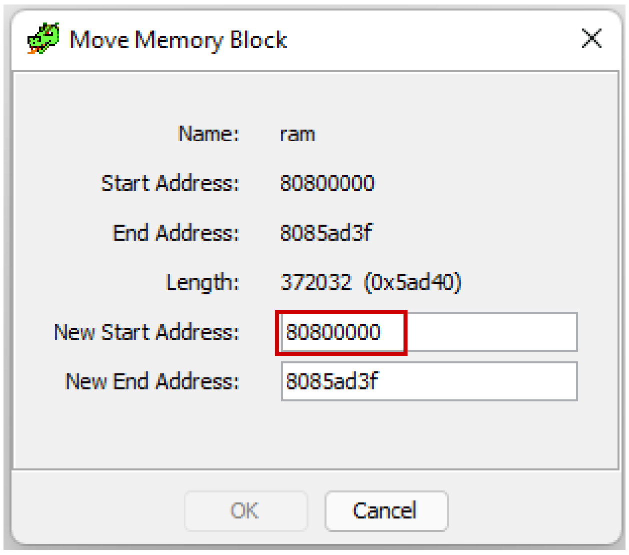
Ghidra loads by default raw binaries at address `0x0` but we can notice that references are highlighted in red because they point outside the binary memory region. We deduce from the references where U-Boot is actually loaded, precisely the offset is `0x0080_0000`.

zi0black: *I must thank @blessthe28, who gave me some clarification regarding the structure of IVT inside a bare metal binary.*



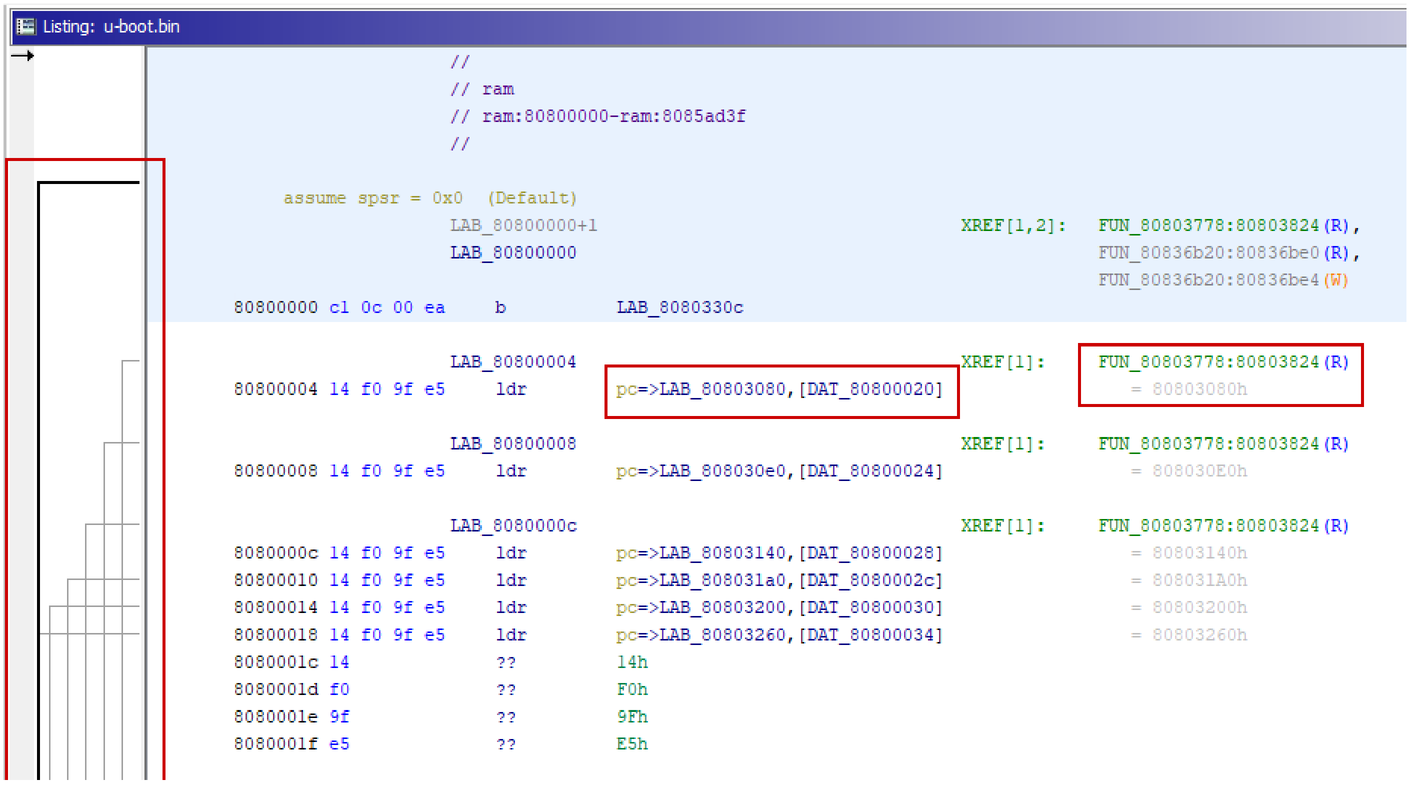
```
                         UndefinedInstruction          XREF[1]:     Entry Point(*)
00000004 14 f0 9f e5     ldr     pc=>LAB_80803080,[DAT_00000020]              = 80803080h
                         SupervisorCall                XREF[1]:     Entry Point(*)
00000008 14 f0 9f e5     ldr     pc=>LAB_808030e0,[DAT_00000024]              = 808030E0h
                         PrefetchAbort                 XREF[1]:     Entry Point(*)
0000000c 14 f0 9f e5     ldr     pc=>LAB_80803140,[DAT_00000028]              = 80803140h
                         DataAbort                     XREF[1]:     Entry Point(*)
00000010 14 f0 9f e5     ldr     pc=>LAB_808031a0,[DAT_0000002c]              = 808031A0h
                         NotUsed                       XREF[1]:     Entry Point(*)
00000014 14 f0 9f e5     ldr     pc=>LAB_80803200,[DAT_00000030]              = 80803200h
                         IRQ                           XREF[1]:     Entry Point(*)
00000018 14 f0 9f e5     ldr     pc=>LAB_80803260,[DAT_00000034]              = 80803260h
                         FIQ                           XREF[1]:     Entry Point(*)
0000001c 14              ??      14h
0000001d f0              ??      F0h
0000001e 9f              ??      9Fh
0000001f e5              ??      E5h
```

Finally, we can relocate the memory block binary inside Ghidra and add the missing memory blocks (i.e. the RAM), this allows Ghidra to better analyze the binary.



```
Move Memory Block

Name:               ram
Start Address:      80800000
End Address:        8085ad3f
Length:             372032 (0x5ad40)
New Start Address:  80800000
New End Address:    8085ad3f
```

After configuring the memory blocks appropriately, we can see that Ghidra correctly identifies all the references!



```
                          //
                          // ram
                          // ram:80800000-ram:8085ad3f
                          //

          assume spsr = 0x0  (Default)
                          LAB_80800000+1                      XREF[1,2]:   FUN_80803778:80803824(R),
                          LAB_80800000                                     FUN_80836b20:80836be0(R),
                                                                           FUN_80836b20:80836be4(W)
     80800000 c1 0c 00 ea     b           LAB_8080330c

                          LAB_80800004                        XREF[1]:     FUN_80803778:80803824(R)
     80800004 14 f0 9f e5     ldr         pc=>LAB_80803080,[DAT_80800020]     = 80803080h

                          LAB_80800008                        XREF[1]:     FUN_80803778:80803824(R)
     80800008 14 f0 9f e5     ldr         pc=>LAB_808030e0,[DAT_80800024]     = 808030E0h

                          LAB_8080000c                        XREF[1]:     FUN_80803778:80803824(R)
     8080000c 14 f0 9f e5     ldr         pc=>LAB_80803140,[DAT_80800028]     = 80803140h
     80800010 14 f0 9f e5     ldr         pc=>LAB_808031a0,[DAT_8080002c]     = 808031A0h
     80800014 14 f0 9f e5     ldr         pc=>LAB_80803200,[DAT_80800030]     = 80803200h
     80800018 14 f0 9f e5     ldr         pc=>LAB_80803260,[DAT_80800034]     = 80803260h
     8080001c 14              ??          14h
     8080001d f0              ??          F0h
     8080001e 9f              ??          9Fh
     8080001f e5              ??          E5h
```

We can now fire Ghidra's auto-analysis with the aggressive search for ARM instructions.

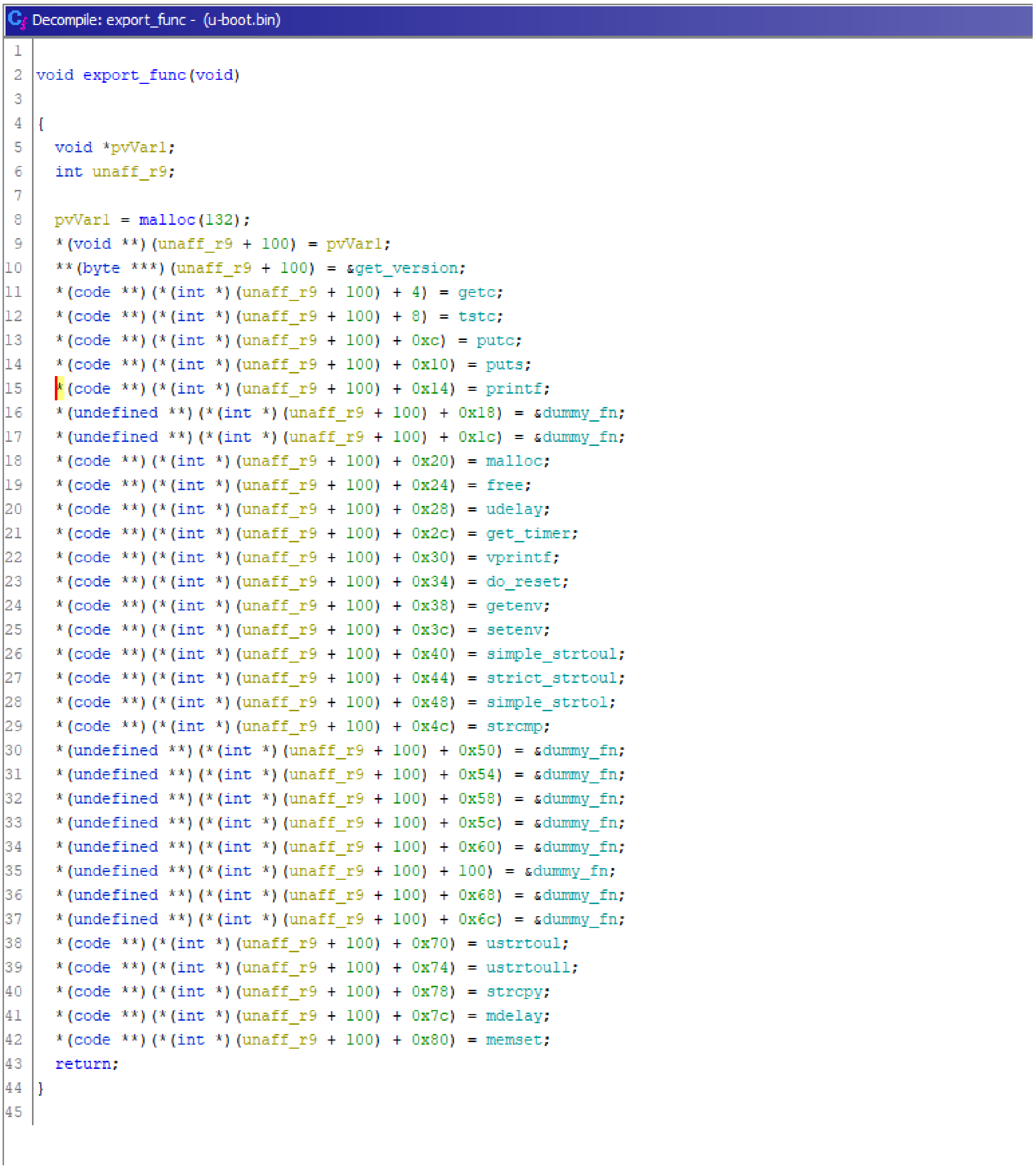
NOTE: sometimes, the aggressive instruction finder defines certain data blocks as code, so pay attention!

Take advantage of open-source

Since we have a custom implementation of U-Boot, we chose to do two things:

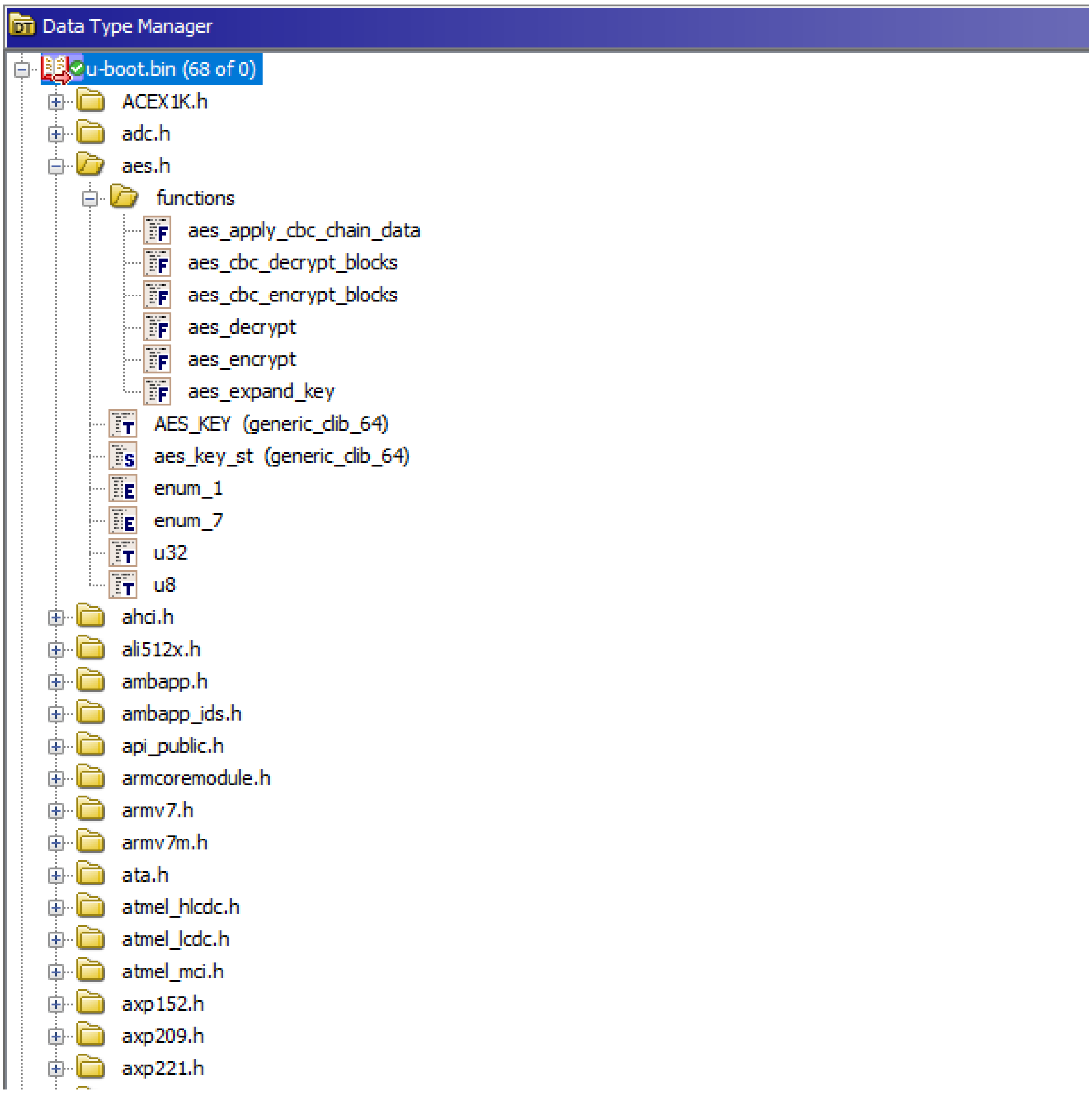
- Define common functions (it's a static binary and doesn't relay to any libC).
- Import U-Boot header files and create a custom DataTypes Library.

So we started reading some source code of *DAS-U-boot* and quickly identified where functions like `memcpy`, `memcp` and `printf` were exported, and then we searched for their code panthers in the binary under-analysis.


Decompile: export_func - (u-boot.bin)

```

void export_func(void)

{
  void *pvVar1;
  int unaff_r9;

  pvVar1 = malloc(132);
  *(void **)(unaff_r9 + 100) = pvVar1;
  **(byte ***)(unaff_r9 + 100) = &get_version;
  *(code **)(*(int *)(unaff_r9 + 100) + 4) = getc;
  *(code **)(*(int *)(unaff_r9 + 100) + 8) = tstc;
  *(code **)(*(int *)(unaff_r9 + 100) + 0xc) = putc;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x10) = puts;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x14) = printf;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x18) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x1c) = &dummy_fn;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x20) = malloc;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x24) = free;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x28) = udelay;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x2c) = get_timer;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x30) = vprintf;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x34) = do_reset;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x38) = getenv;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x3c) = setenv;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x40) = simple_strtoul;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x44) = strict_strtoul;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x48) = simple_strtol;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x4c) = strcmp;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x50) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x54) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x58) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x5c) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x60) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 100) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x68) = &dummy_fn;
  *(undefined **)(*(int *)(unaff_r9 + 100) + 0x6c) = &dummy_fn;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x70) = ustrtoul;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x74) = ustrtoull;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x78) = strcpy;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x7c) = mdelay;
  *(code **)(*(int *)(unaff_r9 + 100) + 0x80) = memset;
  return;
}

```

Meanwhile, we also started building the custom DataTypes Library.
Ghidra has good support for importing header files and resolves other imports automatically. Sometimes you might need to fix the import order if you choose to import only a few header files. This process is far from being an automatic task and requires some handwork to fix of header files.



```
u-boot.bin (68 of 0)
  ACEX1K.h
  adc.h
  aes.h
    functions
      aes_apply_cbc_chain_data
      aes_cbc_decrypt_blocks
      aes_cbc_encrypt_blocks
      aes_decrypt
      aes_encrypt
      aes_expand_key
    AES_KEY (generic_clib_64)
    aes_key_st (generic_clib_64)
    enum_1
    enum_7
    u32
    u8
  ahci.h
  ali512x.h
  ambapp.h
  ambapp_ids.h
  api_public.h
  armcoremodule.h
  armv7.h
  armv7m.h
  ata.h
  atmel_hlcdc.h
  atmel_lcdc.h
  atmel_mci.h
  axp152.h
  axp209.h
  axp221.h
```

We finally have a kinda easy-to-browse project in Ghidra, and we can jump into crypto!

Who doesn’t love crypto reversing? Us!

Now we know exactly the steps for the boot process:

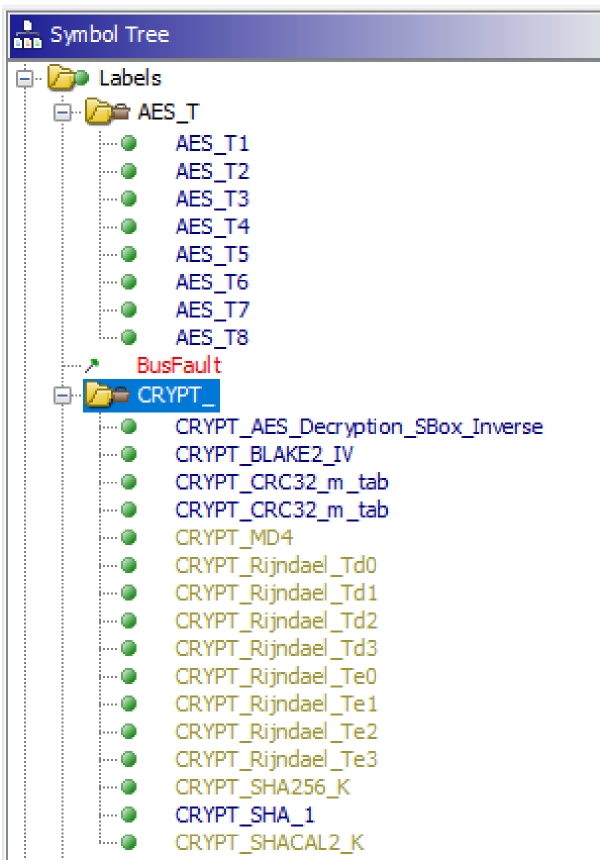
1. The custom U-Boot bootloader is loaded and executed.
2. U-Boot Loads the encrypted Linux kernel in memory.
3. U-Boot perform a key derivation function on some hardcoded data (remember, this device **doesn’t** use ARM Trust Zone).
4. The kernel is decrypted in memory.
5. The bootloader executes the kernel.
6. The kernel *probably* decrypts the rootfs, since the bootloader does not implement such feature.

We still don't know how the decryption mechanism is implemented since it's custom and not present in the U-Boot source code, so we started searching for common cryptographic constants in the code to recognize the cryptographic algorithms that are used.

Basically most of the cryptographic algorithms have some kind of constants that are used to perform various type of operations. For example initialization vectors, seeds, base points, S-Boxes, etc.

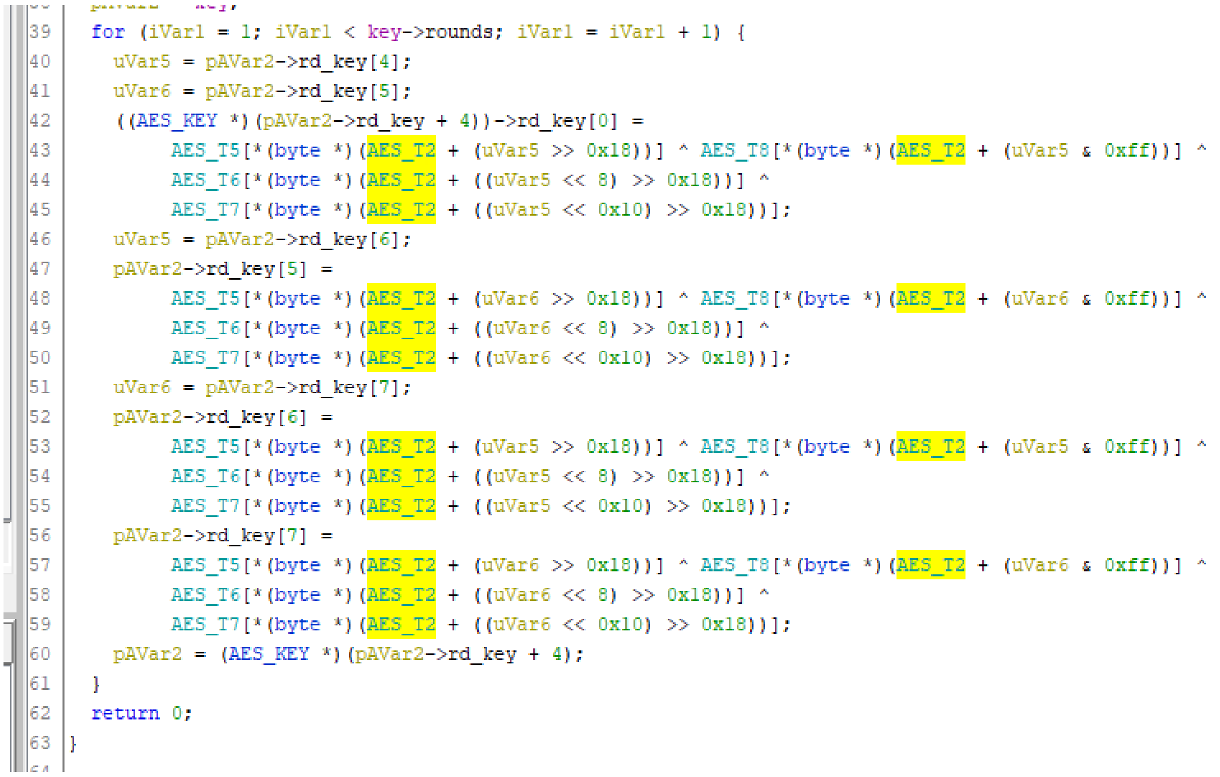
When a cryptographic algorithm is implemented in a programming language, those constants are embedded in the program as data and (in our case) compiled into the binary. So it's possible to search where such constants appear, track the functions that use them and recognize what's going on and which algorithms are being used, even if the binary is stripped.

For this operation we used ghidra-findcrypt on the bootloader binary.



```
Symbol Tree
Labels
  AES_T
    AES_T1
    AES_T2
    AES_T3
    AES_T4
    AES_T5
    AES_T6
    AES_T7
    AES_T8
  BusFault
  CRYPT_
    CRYPT_AES_Decryption_SBox_Inverse
    CRYPT_BLAKE2_IV
    CRYPT_CRC32_m_tab
    CRYPT_CRC32_m_tab
    CRYPT_MD4
    CRYPT_Rijndael_Td0
    CRYPT_Rijndael_Td1
    CRYPT_Rijndael_Td2
    CRYPT_Rijndael_Td3
    CRYPT_Rijndael_Te0
    CRYPT_Rijndael_Te1
    CRYPT_Rijndael_Te2
    CRYPT_Rijndael_Te3
    CRYPT_SHA256_K
    CRYPT_SHA_1
    CRYPT_SHACAL2_K
```

NOTE: ghidra-findcrypt detected a BLAKE2 IV but this is a false positive since BLAKE2b IV is the same as SHA-512 IV, and BLAKE2s IV is the same as SHA-256 IV.



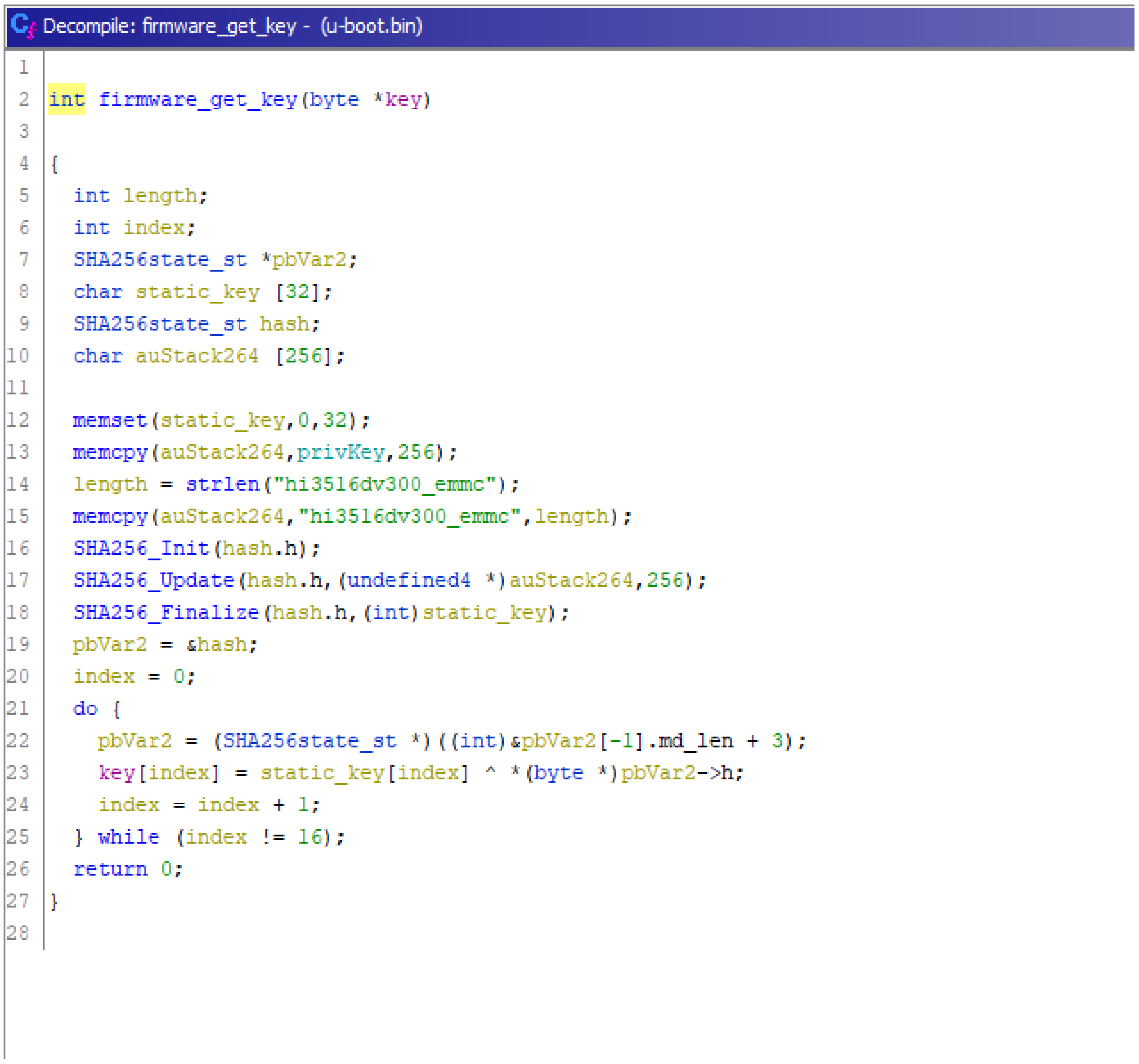
```
  for (iVar1 = 1; iVar1 < key->rounds; iVar1 = iVar1 + 1) {
    uVar5 = pAVar2->rd_key[4];
    uVar6 = pAVar2->rd_key[5];
    ((AES_KEY *)(pAVar2->rd_key + 4))->rd_key[0] =
         AES_T5[*(byte *)(AES_T2 + (uVar5 >> 0x18))] ^ AES_T8[*(byte *)(AES_T2 + (uVar5 & 0xff))] ^
         AES_T6[*(byte *)(AES_T2 + ((uVar5 << 8) >> 0x18))] ^
         AES_T7[*(byte *)(AES_T2 + ((uVar5 << 0x10) >> 0x18))];
    uVar5 = pAVar2->rd_key[6];
    pAVar2->rd_key[5] =
         AES_T5[*(byte *)(AES_T2 + (uVar6 >> 0x18))] ^ AES_T8[*(byte *)(AES_T2 + (uVar6 & 0xff))] ^
         AES_T6[*(byte *)(AES_T2 + ((uVar6 << 8) >> 0x18))] ^
         AES_T7[*(byte *)(AES_T2 + ((uVar6 << 0x10) >> 0x18))];
    uVar6 = pAVar2->rd_key[7];
    pAVar2->rd_key[6] =
         AES_T5[*(byte *)(AES_T2 + (uVar5 >> 0x18))] ^ AES_T8[*(byte *)(AES_T2 + (uVar5 & 0xff))] ^
         AES_T6[*(byte *)(AES_T2 + ((uVar5 << 8) >> 0x18))] ^
         AES_T7[*(byte *)(AES_T2 + ((uVar5 << 0x10) >> 0x18))];
    pAVar2->rd_key[7] =
         AES_T5[*(byte *)(AES_T2 + (uVar6 >> 0x18))] ^ AES_T8[*(byte *)(AES_T2 + (uVar6 & 0xff))] ^
         AES_T6[*(byte *)(AES_T2 + ((uVar6 << 8) >> 0x18))] ^
         AES_T7[*(byte *)(AES_T2 + ((uVar6 << 0x10) >> 0x18))];
    pAVar2 = (AES_KEY *)(pAVar2->rd_key + 4);
  }
  return 0;
}
```

For example, in the image above we can see the AES decryption function after some variable renaming and type definition. Now the code is waaaay more readable than before, and we can diff it against a standard AES decryption function to see if they are the same. And indeed they are!

So we now know that AES is used to decrypt data and SHA1 is used for the key derivation. Our initial idea was to get the encryption keys and function parameters and write a convenient python script to decrypt the kernel.

I bet he's thinking abouth other women.
Who the hell wrote this mode of operation
imgflip.com

Unfortunately, it turned out that it was using a *strange* mode of operation.



```

int firmware_get_key(byte *key)

{
  int length;
  int index;
  SHA256state_st *pbVar2;
  char static_key [32];
  SHA256state_st hash;
  char auStack264 [256];

  memset(static_key,0,32);
  memcpy(auStack264,privKey,256);
  length = strlen("hi3516dv300_emmc");
  memcpy(auStack264,"hi3516dv300_emmc",length);
  SHA256_Init(hash.h);
  SHA256_Update(hash.h,(undefined4 *)auStack264,256);
  SHA256_Finalize(hash.h,(int)static_key);
  pbVar2 = &hash;
  index = 0;
  do {
    pbVar2 = (SHA256state_st *)((int)&pbVar2[-1].md_len + 3);
    key[index] = static_key[index] ^ *(byte *)pbVar2->h;
    index = index + 1;
  } while (index != 16);
  return 0;
}

```

NOTE: The image represents only one of the multiple functions involved in the decryption process and key derivation.

At this point we had two options:

- Ignoring the headache, keep reversing the cryptographic algorithms, and start re-implementing it.
- Getting creative!

As you might guess, we hate headaches, and we love creativity!
In the next blogpost we will explain how we used the information gathered through the reverse engineering process to emulate U-Boot and decrypt the kernel!
STAY TUNED!

zi0black: *A special thanks goes to Enrico `twiz` Perla, the author of "A Guide to Kernel Exploitation", for peer reviewing this blogpost and for being always helpful and kind.*

—

Resources

- http://classweb.ece.umd.edu/enee447.S2016/ARM-Documentation/ARM-Interrupts-3.pdf
- https://www.design-reuse.com/articles/38128/method-for-booting-arm-based-multi-core-socs.html
- https://stackoverflow.com/questions/6139952/what-is-the-booting-process-for-arm
- https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf
- https://interrupt.memfault.com/blog/how-to-write-linker-scripts-for-firmware
- https://www.vinnie.work/docs/embeddedsystemsanalysis/firmware/baremetalbinary/
- https://www.coranac.com/tonc/text/asm.htm
- https://azeria-labs.com/writing-arm-assembly-part-1/

Previous post

QilingLab – Release

Next post

Reversing embedded device bootloader (U-Boot) - p.2