16:07 Extracting the Game Boy Advance BIOS ROM through the Execution of Unmapped Thumb Instructions

by Maribel Hearn

Lately, I've been a bit obsessed with the Game Boy Advance. The hardware is simpler than the modern handhelds I've been playing with and the CPU is of a familiar architecture (ARM7TDMI), making it a rather fun toy for experimentation. The hardware is rather well documented, especially by Martin Korth's GBATEK page.[22] As the GBA is a console where understanding what happens at a cycle-level is important, I have been writing small programs to test edge cases of the hardware that I didn't quite understand from reading alone. One component where I wasn't quite happy with presently available documentation was the BIOS ROM. Closer inspection of how the hardware behaves leads to a more detailed hypothesis of how the ROM protection actually works, and testing this hypothesis turns into the discovery a new method of dumping the GBA BIOS.

[22]http://problemkaputt.de/gbatek.htm

[23]https://mgba.io/2017/06/30/cracking-gba-bios/

Prior Work

Let us briefly review previously known techniques for dumping the BIOS.

The earliest and probably the most well known dumping method is using a software vulnerability discovered by Dark Fader in software interrupt 1Fh. This was originally intended for conversion of MIDI information to playable frequencies. The first argument to the SWI a pointer for which bounds-checking was not performed, allowing for arbitrary memory access.

A more recent method of dumping the GBA BIOS was developed by Vicki Pfau, who wrote an article on the mGBA blog about it,[23] making use of the fact that you can directly jump to any arbitrary address in the BIOS to jump. She also develops a black-box version of the attack that does not require knowledge of the address by deriving what it is at runtime by clever use of interrupts.

But this article is about neither of the above. This is a different method that does not utilize any software vulnerabilities in the BIOS; in fact, it requires neither knowledge of the contents of the BIOS nor execution of any BIOS code.

BIOS Protection

The BIOS ROM is a piece of read-only memory that sits at the beginning of the GBA's address space. In addition to being used for initialization, it also provides a handful of routines accessable by software interrupts. It is rather small, sitting at 16 KiB in size. Games running on the GBA are prevented from reading the BIOS and only code running from the BIOS itself can read the BIOS. Attempts to read the BIOS from elsewhere results in only the last successfully fetched BIOS opcode, so the BIOS from the game's point of view is just a repeating stream of garbage.

This naturally leads to the question: How does the BIOS ROM actually protect itself from improper access? The GBA has no memory management unit; data and prefetch aborts are not a thing that happens. Looking at how emulators implement this

```
                +-------------------+    \
       00000000h|                   |    |
                |BIOS ROM (16 KiB)  |     > Yes, we're interested in this part
       00003FFFh|                   |    |
                +-------------------+    /
       00004000h|Unmapped memory    |
                |                   |
       01FFFFFFh|                   |
                +-------------------+
       02000000h|EWRAM (256KiB)     |
                |On-board work RAM  |
       02FFFFFFh|Mirrored           |
                +-------------------+
       03000000h|IWRAM (32 KiB)     |
                |On-chip Work RAM   |
       03FFFFFFh|Mirrored           |
                +-------------------+
       04000000h|MMIO               |
                |                   |
       040003FFh|                   |
                +-------------------+
       04000400h|Mostly*            |
                |Unmapped Memory    |     *: The I/O port 04000800h alone is mirrored
       04FFFFFFh|                   |        through this region, repeating every 64KiB.
                +-------------------+        (04xx0800h is a mirror of 04000800h.)
       05000000h|Palette RAM        |
                |(1 KiB)            |
       05FFFFFFh|Mirrored           |
                +-------------------+
       06000000h|Video RAM          |     **: Although VRAM is 96KiB = 64KiB + 32KiB,
                |(96 KiB)           |         it is mirrored across memory in blocks of
       06FFFFFFh|Mirrored **        |         128KiB = 64Kib + 32Kib + 32Kib
                +-------------------+         The two 32 KiB blocks are mirrors of
       07000000h|Object Attribute   |         each other.
                |  Memory (OAM)     |
                |(1 KiB)            |
       07FFFFFFh|Mirrored           |
                +-------------------+
       08000000h|Game Pak ROM       |
                |                   |
                |Three mirrors      |
                |with different     |
                |wait states        |
       0DFFFFFFh|                   |
                +-------------------+
       0E000000h|Game Pak SRAM      |
                |(Variable size)    |
                |Mirrored           |
       0FFFFFFFh|                   |
                +-------------------+
       10000000h|Unmapped memory    |
                |                   |
                |                   |
       FFFFFFFFh|                   |   }  Also this part, but spoilers.
                +-------------------+

   GBA Memory Map : Most memory regions are mirrored through each
                    respective memory region, with the exception of
                    the BIOS ROM and MMIO Gaps in the memory map
                    are found after the BIOS ROM, MMIO, and at the
                    end of the address space

                    Diagram based on information from Martin Korth
                    http://problemkaputt.de/gbatek.htm
```

does not help as most emulators look at the CPU's program counter to determine if the current instruction is within or outside of the BIOS memory region and use this to allow or disallow access respectively, but this can't possibly be how the real BIOS ROM actually determines a valid access as wiring up the PC to the BIOS ROM chip would've been prohibitively complex. Thus a simpler technique must have been used.

A normal ARM7TDMI chip exposes a number of signals to the memory system in order to access memory. A full list of them are available in the ARM7TDMI reference manual (page 3-3), but the ones that interest us at the moment are `nOPC` and `A[31:0]`. `A[31:0]` is a 32-bit value representing the address that the CPU wants to read. `nOPC` is a signal that is 0 if the CPU is reading an instruction, and is 1 if the CPU is reading data. From this, a very simple scheme for protecting the BIOS ROM could be devised: if `nOPC` is 0 and `A[31:0]` is within the BIOS memory region, unlock the BIOS. otherwise, if `nOPC` is 0 and `A[31:0]` is outside of the BIOS memory region, lock the BIOS. `nOPC` of 1 has no effect on the current lock state. This serves to protect the BIOS because the CPU only emits a `nOPC=0` signal with `A[31:0]` being an address within the BIOS only it is intending to execute instructions within the BIOS. Thus only BIOS instructions have access to the BIOS.

While the above is a guess of how the GBA actually does BIOS locking, it matches the observed behaviour.

This answers our question on how the BIOS protects itself. But it leads to another: Are there any edge-cases due to this behaviour that allow us to easily dump the BIOS? It turns out the answer to this question is yes.

`A[31:0]` falls within the BIOS when the CPU *intends to* execute code within the BIOS. This does not necessarily mean the code is actually has to be executed, but there only has to be an intent by the CPU to execute. The ARM7TDMI CPU is a pipelined processor. In order to keep the pipeline filled, the CPU accesses memory by prefetching *two instructions ahead* of the instruction it is currently executing. This results in an off-by-two error: While BIOS sits at `0x00000000` to `0x00003FFF`, instructions from two instruction widths ahread of this have access to the BIOS! This corresponds to `0xFFFFFFF8` to `0x00003FF7` when in ARM mode, and `0xFFFF-FFFC` to `0x00003FFB` when in Thumb mode.

Evidently this means that if you could place instructions at memory locations just before the ROM you would have access to the BIOS with protection disabled. Unfortunately there is no RAM backing these memory locations (see GBA Memory Map). This complicates this attack somewhat, and we need to now talk about what happens with the CPU reads unmapped memory.

Executing from Unmapped Memory

When the CPU reads unmapped memory, the value it actually reads is the residual data remaining on the bus left after the previous read, that is to say it is an open-bus read.[24] This makes it simple to make it look like instructions exist at an unmapped memory location: all we need to do is somehow get it on the bus by ensuring it is the last thing to be read from or written to the bus. Since the instruction prefetcher is often the last thing to read from the bus, the value you read from the bus is often the last prefetched instruction.

One thing to note is that since the bus is 32 bits wide, we can either stuff one ARM instruction (1×32 bits) or two Thumb instructions (2×16 bits). Since the first instruction of BIOS is going to be the reset vector at `0x00000000`, we have to do a memory read followed by a return. Thus two Thumb instructions it is.

Where we jump from is also important. Each memory chip puts slightly different things on the bus when a 16-bit read is requested. A table of what each memory instruction places on the bus is shown in Figure 1.

YOUNG BRIGHT COMPUTER PROGRAMMERS WANTED!

To program games, graphics, software etc. on the **Amiga**, **Atari** and **IBM PC compatibles**.

For further details please telephone Michael on

(0252) 877431

or write to:

GAINSTAR

Unit 1, Rear of 7 Wellington Road, Sandhurst, Surrey, GU17 8AW

[24]Does this reliance on the parasitic capacitance of the bus make this more of a hardware attack? Who can say.

```
Values in Memory:
| $-2  | $-1  | $    | $+1  | $+2  | $+3  |
| 0x88 | 0x99 | 0xAA | 0xBB | 0xCC | 0xDD |

Data found on bus after CPU requests 16-bit read of address $.
| Memory Region | Alignment         | Value on bus   |
| ---           | ---               | ---            |
| EWRAM         | doesn't matter    | 0xBBAABBAA     |
| IWRAM         | $ % 4 == 0        | 0x????BBAA (*) |
|               | $ % 4 == 2        | 0xBBAA???? (*) |
| Palette RAM   | doesn't matter    | 0xBBAABBAA     |
| VRAM          | doesn't matter    | 0xBBAABBAA     |
| OAM           | $ % 4 == 0        | 0xDDCCBBAA     |
|               | $ % 4 == 2        | 0xBBAA9988     |
| Game Pak ROM  | doesn't matter    | 0xBBAABBAA     |

(*) IWRAM is rather peculiar. The RAM chip writes to only half of
the bus. This means that half of the penultimate value on the bus
is still visible, here represented by ????.
```

Figure 1. Data on the Bus

Since we want two different instructions to execute, not two of the same, the above table immediately eliminates all options other than OAM and IWRAM. Of the two available options, I chose to use IWRAM. This is because OAM is accessed by the video hardware and thus is only available to the CPU during VBlank and optionally HBlank – this would unnecessarily complicate things.

All we need to do now is ensure that the penultimate memory access puts one Thumb instruction on the bus and that the prefetcher puts the other Thumb instruction on the bus, then immediately jump to the unmapped memory location `0xFFFFFFFC`. Which instruction is placed by what depends on instruction alignment. I've arbitrarily decided to put the final jump on a non-4-byte aligned address, so the first instruction is placed on the bus via a `STR` instruction and the latter is place four bytes after our jump instruction so that the prefetcher reads it. Note that the location to which the `STR` takes place does not matter at all,[25] all we're interested in is what happens to the bus.

By now you ought to see how the attack can be assembled from the ability to execute data left on the bus at any unmapped address, the ability to place two 16-bit Thumb instructions in a single 32-bit bus word, and carefully navigating the pipeline to branch to avoid unmapped instruction and to unlock the BIOS ROM.

[25]Well, if you trash an MMIO register that's your fault really.

BARNES BICYCLES

The Barnes White Flyer

is a Popular and Trustworthy Mount. It can boast of more novel features than any other machine on wheels.

There are OTHER bicycles, but only one BARNES.

=Catalogue Free=

THE BARNES CYCLE CO.,

SYRACUSE, N.Y.

Exploit Summary

Reading the locked BIOS ROM is performed by five steps, which together allow us to fetch one 32-bit word from the BIOS ROM.

1. We put two instructions onto the bus `ldr r0, [r0]; bx lr` (`0x47706800`). As we are starting from IWRAM, we use a store instruction as well as the prefetcher to do this.
2. We jump to the invalid memory address `0xFFFFFFFC` in Thumb mode.[26] The CPU attempts to read instructions from this address and instead reads the instructions we've put on bus.
3. Before executing the instruction at `0xFFFFFFFC`, the CPU prefetches two instructions ahead. This results in a instruction read of `0x00000000` (`0xFFFFFFFC` + 2 * 2). This unlocks the BIOS.
4. Our `ldr r0, [r0]` instruction at `0xFFFFFFFC` executes, reading the unlocked memory.
5. Our `bx lr` instruction at `0xFFFFFFFE` executes, returning to our code.

Assembly

```
.thumb
.section .iwram
.func read_bios, read_bios
.global read_bios
.type read_bios, %function
.balign 4
// u32 read_bios(u32 bios_address):
read_bios:
  ldr r1, =0xFFFFFFFD
  ldr r2, =0x47706800
  str r2, [r1]
  bx r1
  bx lr
  bx lr
.balign 4
.endfunc
.ltorg
```

A new software label setting new standards in Computer Gaming requires **CODERS** and **GRAPHIC ARTISTS** for the **AMIGA, ATARI ST** and **PC MACHINES**, capable of producing the **BEST** the Medium has to offer.

If you think you've got what it takes, contact
ADRIAN TURNER at
ACTUAL SCREENSHOTS
immediately on
(01) 533 2918
or send C.V.'s and DEMOS to:
UNIT 7D, KING'S YARD, CARPENTERS ROAD, LONDON, E15 2HD

Where to store the dumped BIOS is left as an exercise for the reader. One can choose to print the BIOS to the screen and painstakingly retype it in, byte by byte. An alternative and possibly more convenient method of storing the now-dumped BIOS - should one have a flashcart — could be storing it to Game Pak SRAM for later retrieval. One may also choose to write to another device over SIO,[27] which requires a receiver program (appropriately named `recver`) to be run on an attached computer.[28] As an added bonus this technique does not require a flashcart as one can load the program using the GBA's multiboot protocol over the same cable.

– – – –   – – –   – – – –

This exploit's performance could be improved, as `ldr r0, [r0]` is not the most efficient instruction that can fit. `ldm` would retrieve more values per call.

Could this technique apply to the ROM from other systems, or perhaps there is some other way to abuse our two primitives: that of data remaining on the bus for unmapped addresses and that of the unexecuted instruction fetch unlocking the ROM?

Acknowledgments

Thanks to Martin Korth whose documentation of the GBA proved invaluable to its understanding. Thanks also to Vicki Pfau and to Byuu for their GBA emulators which I often reference.

[26] This appears in the assembly as a branch to `0xFFFFFFFD` because the least significant bit of the program counter controls the mode. All Thumb instructions are odd, and all ARM instructions are even.

[27] `unzip pocorgtfo16.pdf iodump.zip`

[28] `git clone https://github.com/MerryMage/gba-multiboot`

Instruction	Cycle*	PC	What's happening	A[31:0]	nOPC	Bus contents	
`str r2, [r1]`	1	`read_bios+4`	Prefetch of read_bios+8	`read_bios+8`	0	`[read_bios+8]`	read
	2	`read_bios+4`	Data store of 0x68006800	`0xFFFFFFFD`	1	`0x68006800`	write
`bx r1`	1	`read_bios+8`	Prefetch of read_bios+10	`read_bios+10`	0	`0x47706800`	read
	2	`read_bios+8`	Pipeline reload (0x6800 is read into pipeline)	`0xFFFFFFFC`	0	`0x47706800`	read
	3	`read_bios+8`	Pipeline reload (0x4770 is read into pipeline)	`0xFFFFFFFE`	0	`0x47706800`	read
`ldr r0, [r0]`	1	`0xFFFFFFFC`	Prefetch of 0x00000000	`0x00000000`	0	`[0x00000000]`	read
	2	`0xFFFFFFFC`	Data read of [r0]	`r0`	1	`[r0]`	read
`bx lr`	1	`0xFFFFFFFE`	Prefetch of 0x00000002	`0x00000002`	0	`[0x00000002]`	read
	2	`0xFFFFFFFE`	Pipeline reload	`lr`	0	`[lr]`	read
	3	`0xFFFFFFFE`	Pipeline reload	`lr+2`	0	`[lr+2]`	read
		`lr`					

Figure 2. Cycle Counts, Excluding Wait States