Duo Labs Report

Examining Personal Protection Devices

Hardware & Firmware Research Methodology in Action

DUO LABS

CISCO Duo Security is now part of Cisco.

Examining Personal Protection Devices

Hardware & Firmware

Research Methodology in Action

Table of Contents

AUTHOR

Todd Manning

DESIGNER

Chelsea Lewis

VERSION

1.0

PUBLISHED

XX/XX/2017

1.0

Introduction

The Duo Labs research team recently examined the Revolar Instinct and the Athena by ROAR for Good, two “panic-button” style personal protection devices that work with accompanying mobile apps on a user’s phone to quickly send alerts to pre-configured contacts. The devices let the user communicate their status without retrieving their phone, unlocking it and sending text messages or making calls.

Our inspection of the hardware and firmware on these devices proved both challenging and interesting. The nuances and idiosyncrasies of each of the personal protection products we examined required a flexible and creative research approach along with a fair bit of patience and determination. The Revolar Instinct, for example, allows local users to download device firmware while ROAR’s Athena device made things more challenging by using readback protection in an attempt to prevent firmware access. Our defeat of this readback protection using an updated method based on prior research makes up a significant part of the work presented here.

Beyond the ability to circumvent the Athena’s firmware readback protection, we found no significant vulnerabilities in either unit’s hardware, firmware or over-the-air (OTA) update mechanisms. Our research, however, did require a foray into the world of ARM Cortex processors and the defeat of the aforementioned Nordic Semiconductor readback protection -- two complex efforts worthy of a methodology paper on their own.

Any researcher tasked with examining or reverse engineering similar devices can benefit from the methods and tactics described in detail here. From teardown to component identification to thorough retrieval and examination of the firmware, we share all of the tools and techniques we used to dig into these increasingly popular devices. We even show you our missteps and do-overs along the way so you can learn along with us.

Summary of Findings

Our hardware and firmware deep dive into the Revolar Instinct and Athena personal protection devices gave us several useful insights, chief among them:

- Creative refinement of prior research and meticulous identification of program instruction addressing allowed us to examine even purposely-hardened firmware with readback protection enabled.

- This technique has broad applicability for security researchers examining the growing number of embedded systems and Internet of Things (IoT) devices.

- Our personal protection device research also led to the development of two IDAPython modules useful for discovering code and functions and annotating vectors tables in similar ARM-based embedded systems.

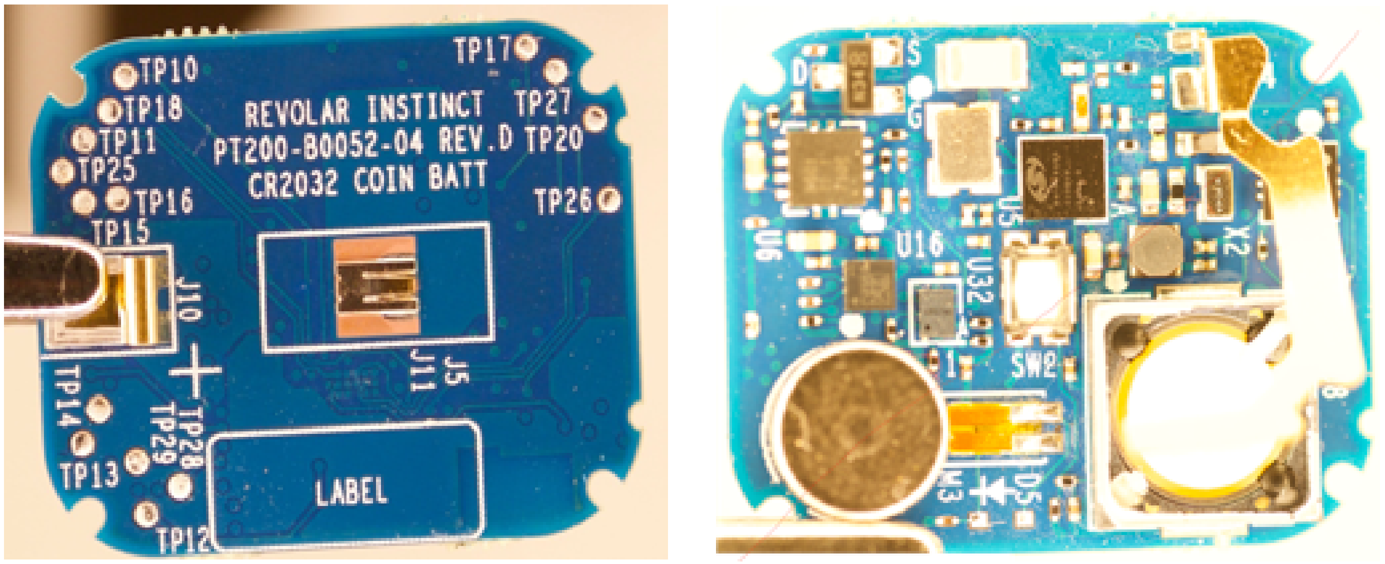
- Duo Labs researchers will continue to develop and share these Python modules to provide improved firmware analysis tools to the security research community at large.

2.0

Technical Overview

Examining the Revolar Instinct

My analysis of Denver-based Revolar's Instinct device began with a teardown of the hardware and photographing the printed circuit board (PCB) to identify components and connections. The images below show the two sides of the board. On the left, the upper side of the board shows the components comprising the device. On the right, the lower side reveals a number of test points that can be used to measure and interact with the various components.



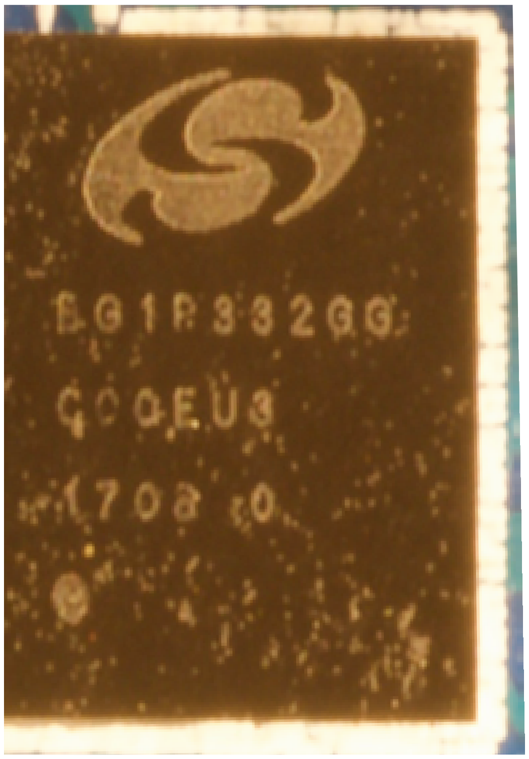
The upper and lower side of the Revolar Instinct PCB

Zooming in on the Revolar's components reveals identifying chip markings. As a researcher, there are a number of ways to use such chip markings to identify key parts. Some of this comes from your standard search engine or by scanning the offerings from your favorite hardware component resellers. After doing this type of work for a while, you start to recognize manufacturer part names and logos on sight. Once you identify the part, the datasheet is your next stop.

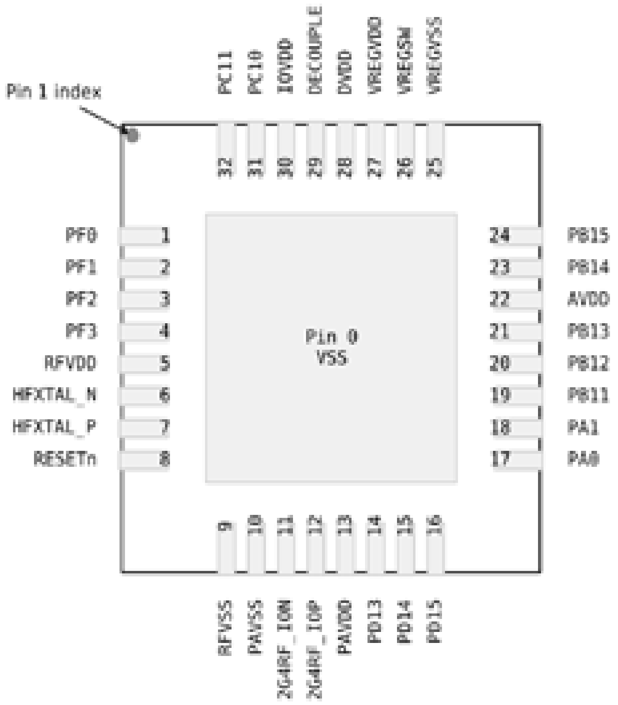
As shown in the image below, the markings on the Revolar microcontroller start with the string **BG1P332GG**, while the logo at the top indicates the part is manufactured by Silicon Labs.

Searching for the part number shows the Revolar is based on a Silicon Labs Blue Gecko line[1] of Bluetooth system-on-chip (SoC) devices. These devices are ARM Cortex M4 devices with 256k of flash program memory and 32k of RAM. I located the Silicon Labs datasheet for the product, and translated the chip markings into a specific part number. Finding a datasheet for a target device gives the researcher necessary information for proceeding with a hardware audit. Using the part number allows the researcher to determine the features present in a particular chip model.

The Silicon Labs SoC present in the Revolar device allows debugging using the serial wire debug (SWD) interface. SWD is a two-pin variant of the more popular four-pin Joint Test Action Group (JTAG) debugging interface. The datasheet identifies the pins assigned to the SWD interface. The SWD clock pin is located on PF0, and the SWDIO pin is located on pin PF1. The pin layout is shown in the image below.



Chip markings reveal the Silicon Labs BG1P332GG

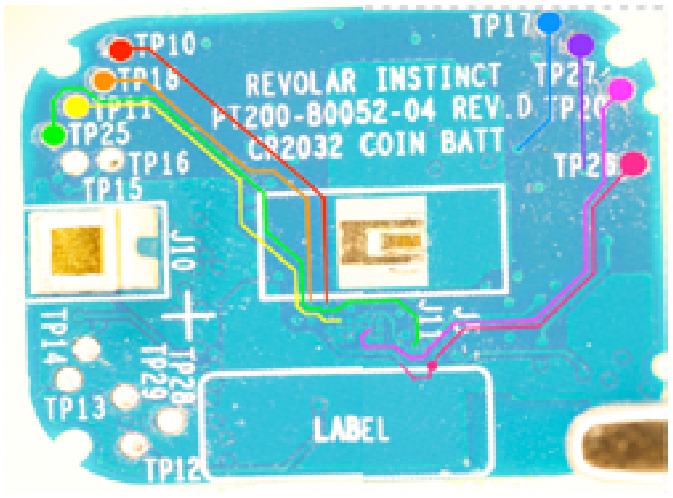


Pinout diagram from the Silicon Labs datasheet

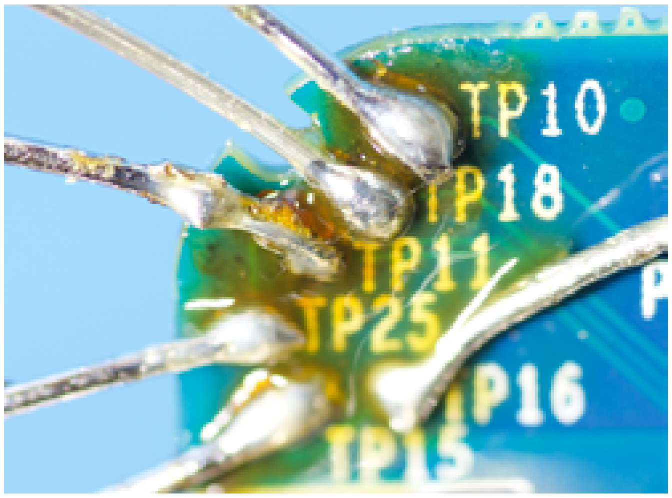
[1] https://www.silabs.com/documents/login/data-sheets/efr32bg1-datasheet.pdf

Identification of the pins is a good first step toward attaching a hardware debug interface and retrieving firmware. The Revolar fortunately provides several test points on the lower side of the board. I performed manual analysis of the traces going to all of the test points in order to identify those that connect to the Silicon Labs SoC. The following test points appear to be routed to the target component:

- **TP10**
- **TP11**
- **TP18**
- **TP20**
- **TP25**
- **TP26**



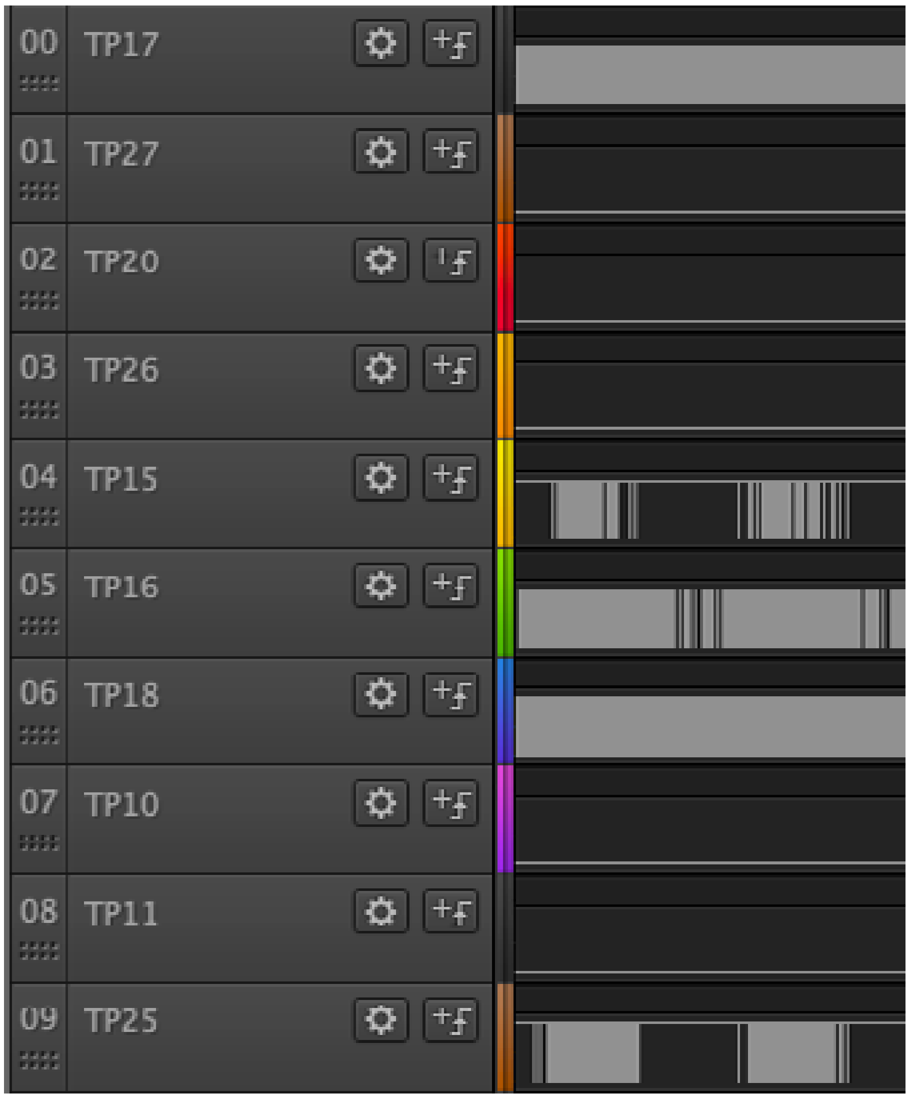
Manually drawing PCB traces to find likely debugging candidate test pads



Leads connected to PCB test pads

I identified the debugging pins by connecting lead wires to the possible test points identified above. Don't judge my soldering skills too harshly. As the following image shows, connecting leads even to well-marked test points can be difficult with a shaky hand.

With the leads connected to the board, I connected a Saleae logic analyzer to all the chips and performed a capture while booting the device. The following capture shows activity on several pins right from startup.



Using Saleae logic analyzer on Revolar Instinct test pads

This image shows the voltages sampled off each pin. Several lines are steadily low, like TP27, TP20, TP26, TP10, and TP11. Test points TP17 and TP18 might be clock signals. With SWD we know there are two pins involved; one pin is a clock while the other is a data line for input and output. With this in mind, I connected a hardware debugger to the pins in an attempt to connect to the SWD interface. I determined that test points TP10 and TP18 are connected to the debugging interface pins.

The following excerpt shows a successful connection to the device using the Segger JLink debugging interface.

```
$ JLinkExe -device cortex-m4 -if swd -speed 4000 -autoconnect 1

Connecting to target via SWD
AHB-AP ROM: 0xE00FF000 (Base addr. of first ROM table)
CPUID reg: 0x410FC241. Implementer code: 0x41 (ARM)
Found Cortex-M4 r0p1, Little endian.
FPUnit: 6 code (BP) slots and 2 literal slots
Cortex-M4 identified.
```

From here I can extract the flash contents and begin firmware analysis. I used the JLink tools distributed by Segger for debugging the device. In particular, I used JLink Commander to extract memory from the device. JLlink Commander is similar to debuggers like GNU Debugger (GDB), allowing the user to set breakpoints and watchpoints, halt and resume execution and examine memory and registers. The debugger supports a command savebin that will transfer the contents of specified memory locations to a local file. This command can be used to extract the firmware from the device for analysis.

Many ARM Cortex devices can implement an optional Memory Protection Unit (MPU) that can protect memory from being read or written. Silicon Labs documentation indicates the MPU can control access to regions of memory. Two execution contexts are supported, User and Privileged. The MPU API allows control over read, write and execute permissions for regions of memory. However, the Revolar Instinct does not appear to use read protection to prevent reading the firmware.

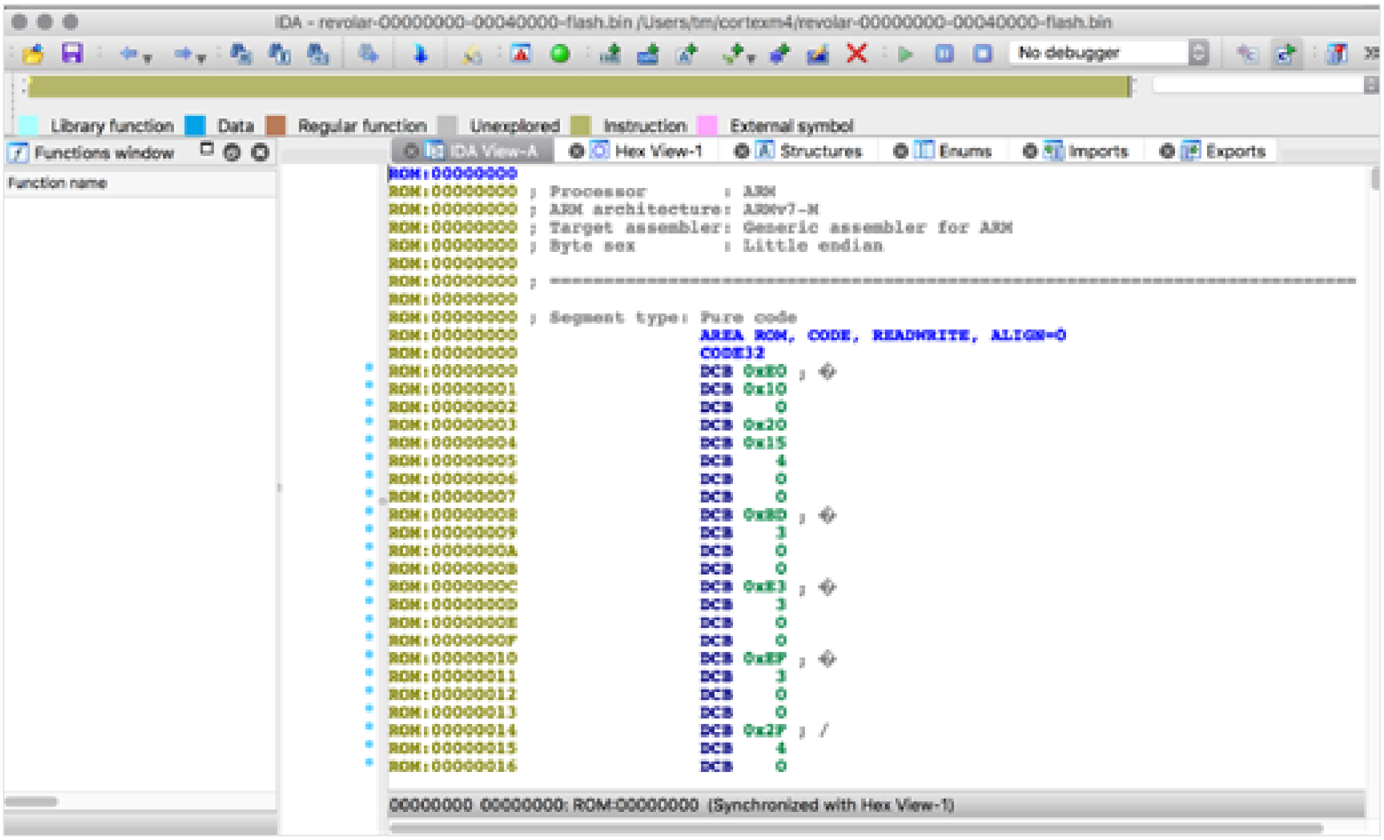
Revolar Instinct Firmware

Knowledge about the device hardware is paramount to understanding the firmware. Knowledge of the processor architecture is required in order to perform correct disassembly of the bytes extracted from the device over SWD.

I performed reverse engineering using IDA Pro and Capstone. When loading firmware into IDA Pro, reverse engineers must set the proper processor type, base architecture, endianness and any other relevant options (such as the use of ARM Thumb instructions). The Silicon Labs data sheet for this chip indicates the EFR32BG1 is an ARM Cortex M4 using little endian byte ordering. For ARM Cortex M4, the appropriate base architecture is ARMv7-M.

Importing Revolar Instinct Firmware into IDA Pro

After setting the appropriate processor features for this firmware, we find that IDA Pro doesn't know much more about the firmware image than what we've configured. In the screenshot below, we see that IDA considers the firmware to be essentially a list of bytes. IDA provides a color-coded legend just beneath the tool bar that indicates the entire contents of the file are considered data, not code.



```
ROM:00000000
ROM:00000000 ; Processor         : ARM
ROM:00000000 ; ARM architecture: ARMv7-M
ROM:00000000 ; Target assembler: Generic assembler for ARM
ROM:00000000 ; Byte sex          : Little endian
ROM:00000000
ROM:00000000 ; ===========================================================================
ROM:00000000
ROM:00000000 ; Segment type: Pure code
ROM:00000000                 AREA ROM, CODE, READWRITE, ALIGN=0
ROM:00000000                 CODE32
ROM:00000000                 DCB 0xE0 ; �
ROM:00000001                 DCB 0x10
ROM:00000002                 DCB    0
ROM:00000003                 DCB 0x20
ROM:00000004                 DCB 0x15
ROM:00000005                 DCB    4
ROM:00000006                 DCB    0
ROM:00000007                 DCB    0
ROM:00000008                 DCB 0xBD ; �
ROM:00000009                 DCB    3
ROM:0000000A                 DCB    0
ROM:0000000B                 DCB    0
ROM:0000000C                 DCB 0xE3 ; �
ROM:0000000D                 DCB    3
ROM:0000000E                 DCB    0
ROM:0000000F                 DCB    0
ROM:00000010                 DCB 0xEF ; �
ROM:00000011                 DCB    3
ROM:00000012                 DCB    0
ROM:00000013                 DCB    0
ROM:00000014                 DCB 0x2F ; /
ROM:00000015                 DCB    4
ROM:00000016                 DCB    0
```

```
00000000 00000000: ROM:00000000 (Synchronized with Hex View-1)
```

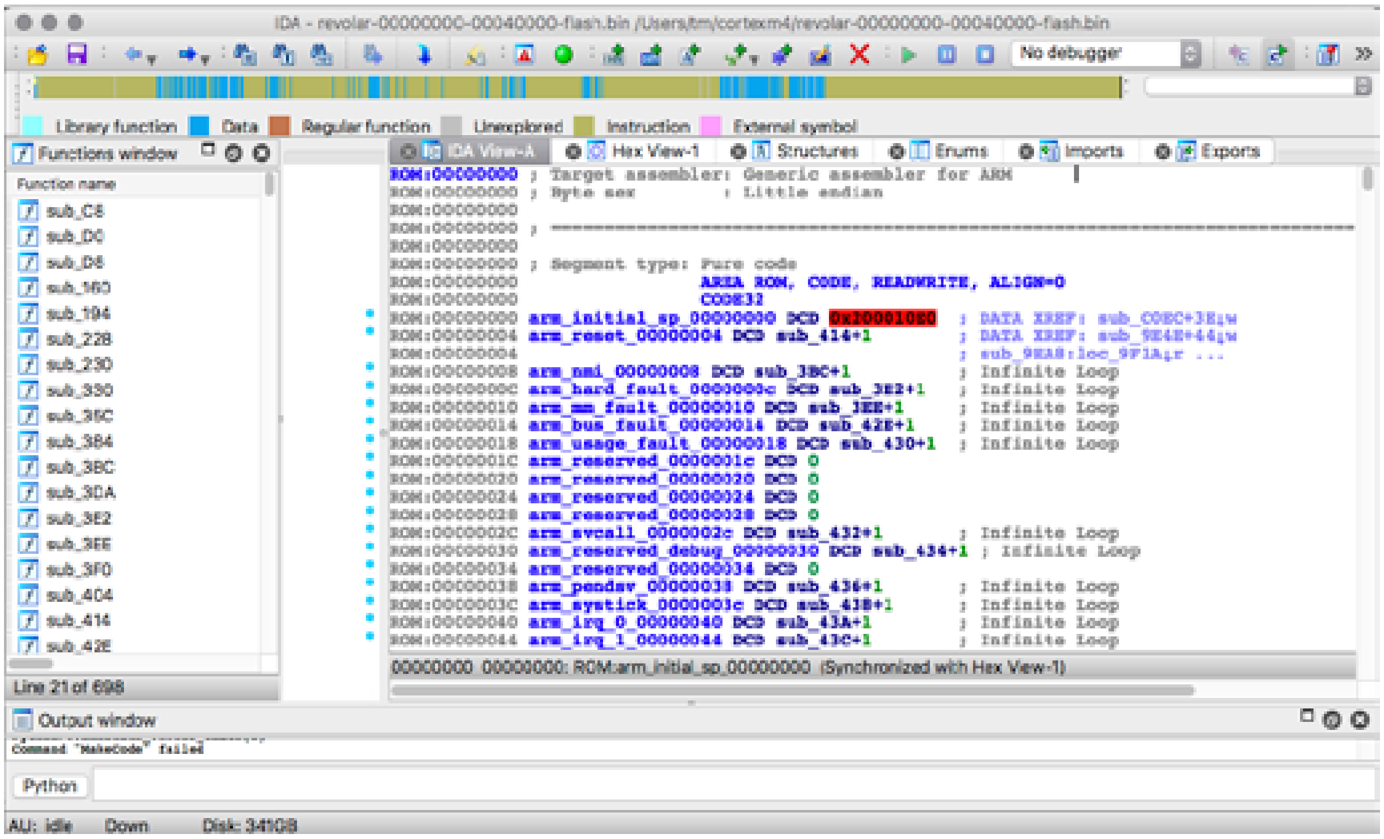
IDA Pro after initial load of ARM Cortex firmware indicating no code defined in database

Understanding of the file layout of ARM Cortex M firmware will go a long way to taking this IDA database from a giant blob of data into something more useful. The goal here is to identify all the code present in this firmware in order to start understanding how it works. I familiarized myself with the ARM Cortex M documentation.

The starting point in analysis of Cortex M firmware is to find the vector table. This table is a collection of four byte values. The first four bytes are the initial stack pointer (InitialSP) value. These four bytes, stored in little endian form, are copied into the microcontroller's stack pointer (SP) register on reset. After this InitialSP value, the remainder of the table contains vectors which are function pointers. The ARM Cortex M4 documentation defines the first 16 vectors. The Reset vector is executed when the device is powered up.

Exception Number	Exception
1	Reset
2	NMI
3	HardFault
4	MemManage
5	BusFault
6	UsageFault
7-10	Reserved
11	SVCall
12	DebugMonitor
13	Reserved
14	PendSV
15	SysTick
16	External interrupt 0

With this knowledge, we can start firmware analysis by annotating this table in IDA. Since these are function pointers, it makes sense to mark the locations in the firmware to which each of these vectors point as code. The following shows the results of giving IDA hints about the functions indicated by the vector-table pointers.



```
sub_C8
sub_D0
sub_D8
sub_160
sub_194
sub_228
sub_230
sub_330
sub_35C
sub_384
sub_3BC
sub_3DA
sub_3E2
sub_3EE
sub_3F0
sub_404
sub_414
sub_42E
```

```
ROM:00000000 ; Target assembler: Generic assembler for ARM
ROM:00000000 ; Byte sex          : Little endian
ROM:00000000
ROM:00000000 ; ===========================================================================
ROM:00000000
ROM:00000000 ; Segment type: Pure code
ROM:00000000                 AREA ROM, CODE, READWRITE, ALIGN=0
ROM:00000000                 CODE32
ROM:00000000 arm_initial_sp_00000000 DCD 0x20001080    ; DATA XREF: sub_C0BC+3E↓w
ROM:00000004 arm_reset_00000004 DCD sub_414+1          ; DATA XREF: sub_9E4E+44↓w
ROM:00000004                                           ; sub_9EA8:loc_9F1A↓r ...
ROM:00000008 arm_nmi_00000008 DCD sub_3BC+1            ; Infinite Loop
ROM:0000000C arm_hard_fault_0000000c DCD sub_3E2+1     ; Infinite Loop
ROM:00000010 arm_mm_fault_00000010 DCD sub_3EE+1       ; Infinite Loop
ROM:00000014 arm_bus_fault_00000014 DCD sub_42E+1      ; Infinite Loop
ROM:00000018 arm_usage_fault_00000018 DCD sub_430+1    ; Infinite Loop
ROM:0000001C arm_reserved_0000001c DCD 0
ROM:00000020 arm_reserved_00000020 DCD 0
ROM:00000024 arm_reserved_00000024 DCD 0
ROM:00000028 arm_reserved_00000028 DCD 0
ROM:0000002C arm_svcall_0000002c DCD sub_432+1         ; Infinite Loop
ROM:00000030 arm_reserved_debug_00000030 DCD sub_434+1 ; Infinite Loop
ROM:00000034 arm_reserved_00000034 DCD 0
ROM:00000038 arm_pendsv_00000038 DCD sub_436+1         ; Infinite Loop
ROM:0000003C arm_systick_0000003c DCD sub_438+1        ; Infinite Loop
ROM:00000040 arm_irq_0_00000040 DCD sub_43A+1          ; Infinite Loop
ROM:00000044 arm_irq_1_00000044 DCD sub_43C+1          ; Infinite Loop
00000000 00000000: ROM:arm_initial_sp_00000000 (Synchronized with Hex View-1)
```

```
Command "MakeCode" failed
```

IDA Pro after annotating the vector table — new regions of code identified

As you can see, the IDA Pro legend of this database indicates multiple locations containing code. In this legend, blue is code. Code is good. The ARMv7-M Architecture Reference Manual documents a minimum of 16 vectors in the vector table. It's important to note that while ARM creates processor specifications, implementers like Silicon Labs have the freedom to expand the vector table beyond the basic 16 entries. The EFR32BG1 datasheet indicates this particular chip can have up to 31 GPIO inputs supporting asynchronous interrupts. This matched what I found in this firmware image. Our vector table annotation kicked off IDA Pro to perform analysis of the code and IDA discovered a function defined just after the vector table, which gives confidence as to the correctness of this approach.

Firmware Analysis on Cortex M Microcontrollers

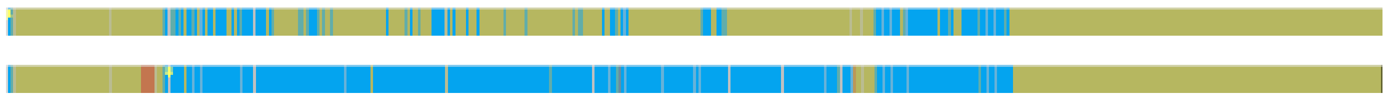
I continued reversing the firmware manually, trying to determine if particular bytes should be converted to code or left defined as data in IDA. The goal is to convert as much of that bar to blue as possible, but it's not as simple as converting all the bytes to code. ARM code often uses program counter (PC)-relative offsets when representing 32-bit values. These values exist in the bytes between functions.

I discovered and even crowd-sourced a few methods for finding the remaining code in the firmware. The first method is to look for the BX LR instruction, which is a common way for ARM functions to return to the calling function. This instruction is encoded in Thumb mode as the two-byte value 0x4770. This proved a successful heuristic as nearly 700 matching instructions exist in the firmware.

The successful search for return instructions got me thinking about other common kinds of ARM instructions that start and end functions. The first few instructions to begin and end functions are named function prologues and function epilogues. Different compilers handle these differently, but there are a number of instructions that are common. Pushing and popping register values in and out of the stack is a common pattern in the functions observed at this point of the firmware disassembly. ARM Cortex M supports both ARM mode and Thumb mode. The heuristic for finding ARM code should cover both instruction sets. Applying these heuristics to the firmware yielded great results.

The first image shows the IDA legend after annotating the vector table, and the second shows the legend after applying the instruction-detection heuristics. The amount of additional code discovered is considerable; triple the number of functions identified as compared to annotating the vector table alone.

I developed two IDA Python modules to implement both the Cortex M firmware importing and ARM code heuristic detection (See Appendix B). The code for these modules is available on Github.



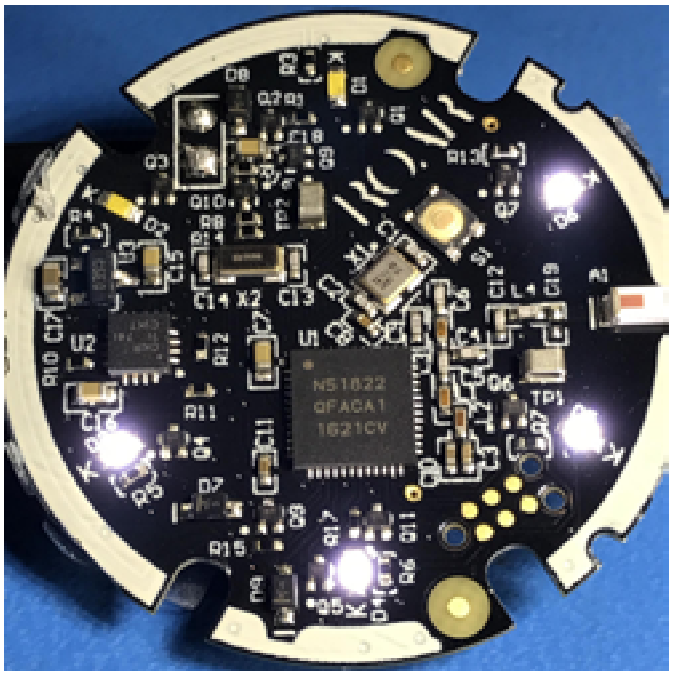
A comparison of code detected after running IDAPython modules against Revolar Instinct firmware

Examining ROAR for Good's Athena

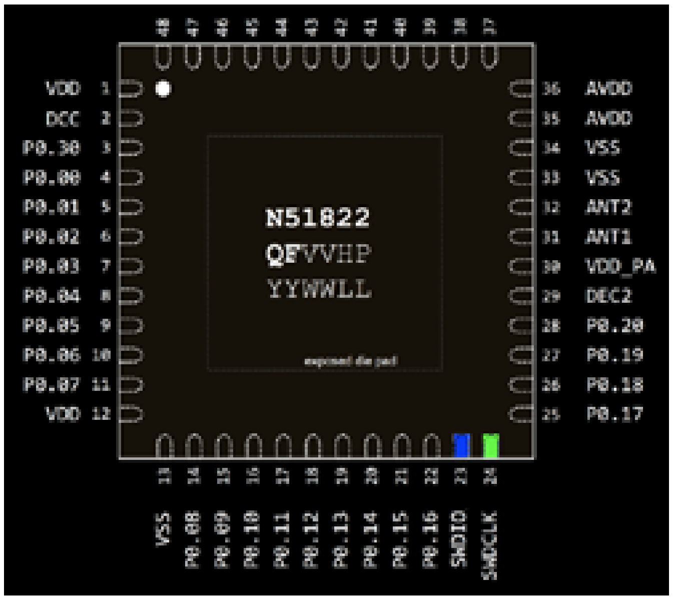
Duo Labs also investigated the Athena personal protection device made by Philadelphia-based ROAR for Good. Again, identification of the target hardware was the first step.

The Athena's microcontroller is a Nordic Semiconductor nRF51822.[2] This is the largest chip near the center of the board. The second row of chip markings indicate this chip contains 256k of code space and 32k of RAM. The presence of a set of exposed test points in the lower right is a welcome sight.

Referencing the datasheet for this Nordic Semiconductor SoC, you can see there are pins that appear to be dedicated to the SWD debugging interface. From the following diagram, you can see that pins 23 and 24 are SWDIO and SWDCLK, respectively. These are the two pins required to attach your debug interface to the chip.



Athena upper side, showing Nordic Semiconductor BLE SoC



Nordic Semiconductor datasheet showing device pinout

[2] http://infocenter.nordicsemi.com/pdf/nRF51_RM_v1.1.pdf

With the goal of connecting to SWD in mind, and that obvious set of exposed test pads next to the chip, I used my multimeter to test connectivity from the target SWD pins to the test pads. I was happy to see two of the pads are indeed connected to the target pins. The photograph below, altered in an image editor to increase contrast, clearly shows the pins and test points.

I soldered wires to the test pads and connected them to the Segger JLink debugger to extract firmware. I was able to communicate with the chip using JLink Commander, but before I got far, the wires I had soldered to the test pads pulled completely off the board taking the pads with them. A few attempts at reconnecting the leads ended with a successful connection directly to the solder joints connecting the QFN48 package to the board.

I ended up having to perform this procedure twice, as these tiny, 32 AWG leads broke off the first time. Hat-tip to fellow Duo Labs researcher Mikhail Davidov for suggesting hot gluing the wires to the board the second time. With the device now wired up, I was able to pull the firmware for analysis.

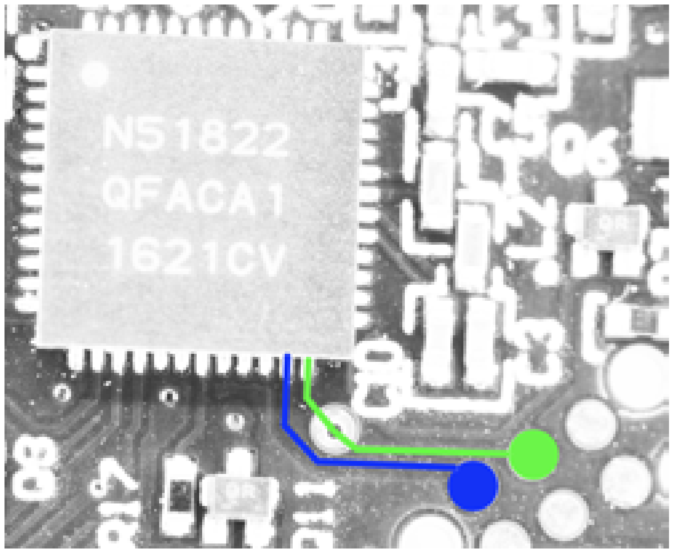
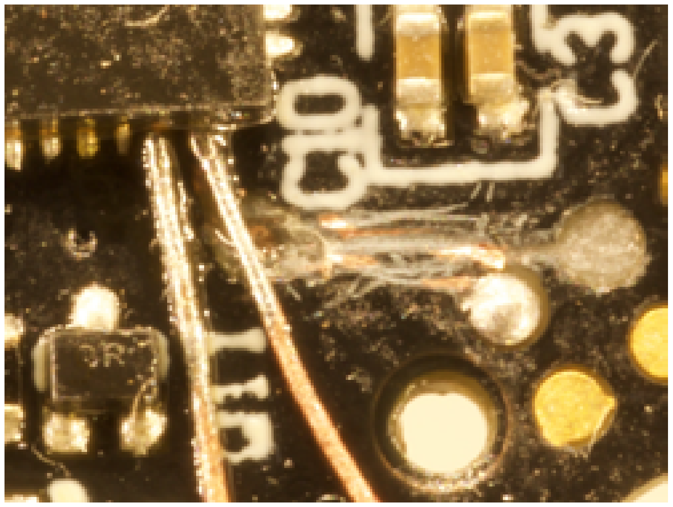


Image of Athena showing debug pins broken out on test pads



Athena debug leads soldered directly to chip after damage to test pads

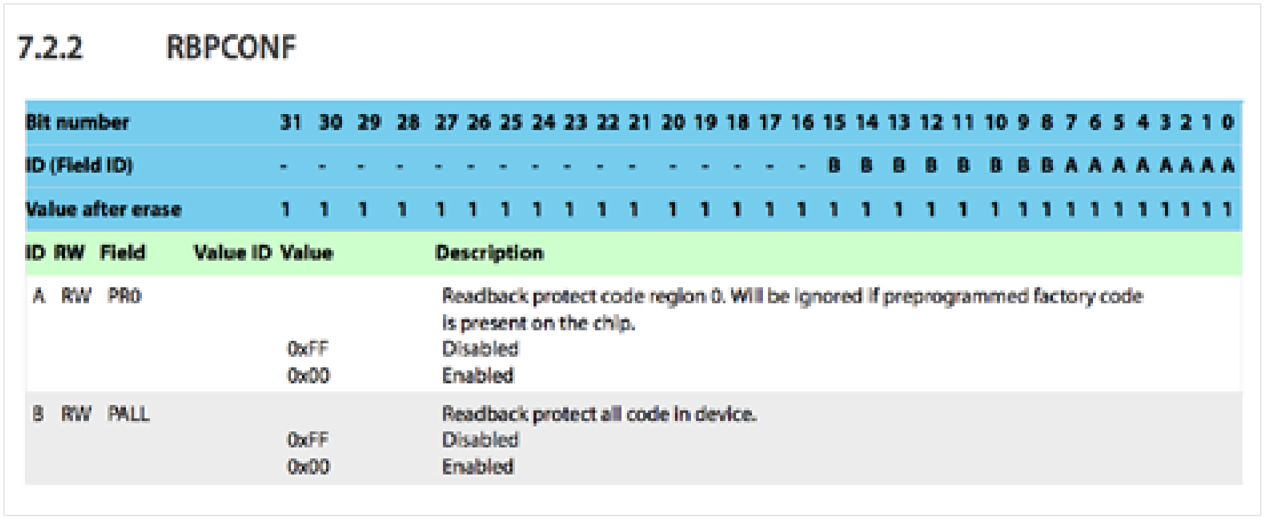
Athena Firmware Extraction

Once I had the SWD interface wired into my JLink debugger, I set out to extract the firmware for analysis just as I'd done with the Revolar device. Using JLink Commander, I attempted to perform a memory read to inspect the contents of the nRF51822 flash storage.

```
$ JLinkExe -device cortex-m0 -if swd -speed 4000 -autoconnect 1
...
J-Link>mem32 0 100
00000000 = 00000000 00000000 00000000 00000000
00000010 = 00000000 00000000 00000000 00000000
00000020 = 00000000 00000000 00000000 00000000
00000030 = 00000000 00000000 00000000 00000000
0000
```

This result indicates the nRF51822 in ROAR's Athena has readback protection enabled. I believed this to be true because the data returned from address 0x0 contained only NULL bytes. As discussed previously, ARM Cortex firmware images begin with a vector table comprised of an initial stack pointer value, the reset handler pointer and a number of other vectors that point to exception-handling functions. Their absence when reading memory is a good indication that readback protection is enabled.

Research into the nRF51822 showed readback protection is enabled by writing to a memory mapped register in the appropriate User Information Configuration Register (UICR). The readback protection configuration (RBPCONF) register is at address 0x10001004. The UICR is a 32-bit value. The following documentation excerpt shows how the microcontroller interprets the UICR.RBPCONF fields.



7.2.2 RBPCONF

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID (Field ID)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A
Value after erase	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

ID	RW	Field	Value ID	Value	Description
A	RW	PR0			Readback protect code region 0. Will be ignored if preprogrammed factory code is present on the chip.
				0xFF	Disabled
				0x00	Enabled
B	RW	PALL			Readback protect all code in device.
				0xFF	Disabled
				0x00	Enabled

Nordic Semiconductor documentation describing readback protection register

Using JLink Commander, I read UICR.RBPCONF from address 0x10001004. The value returned was 0xFFFF00FF. As you can see from the documentation, this value corresponds to the following:

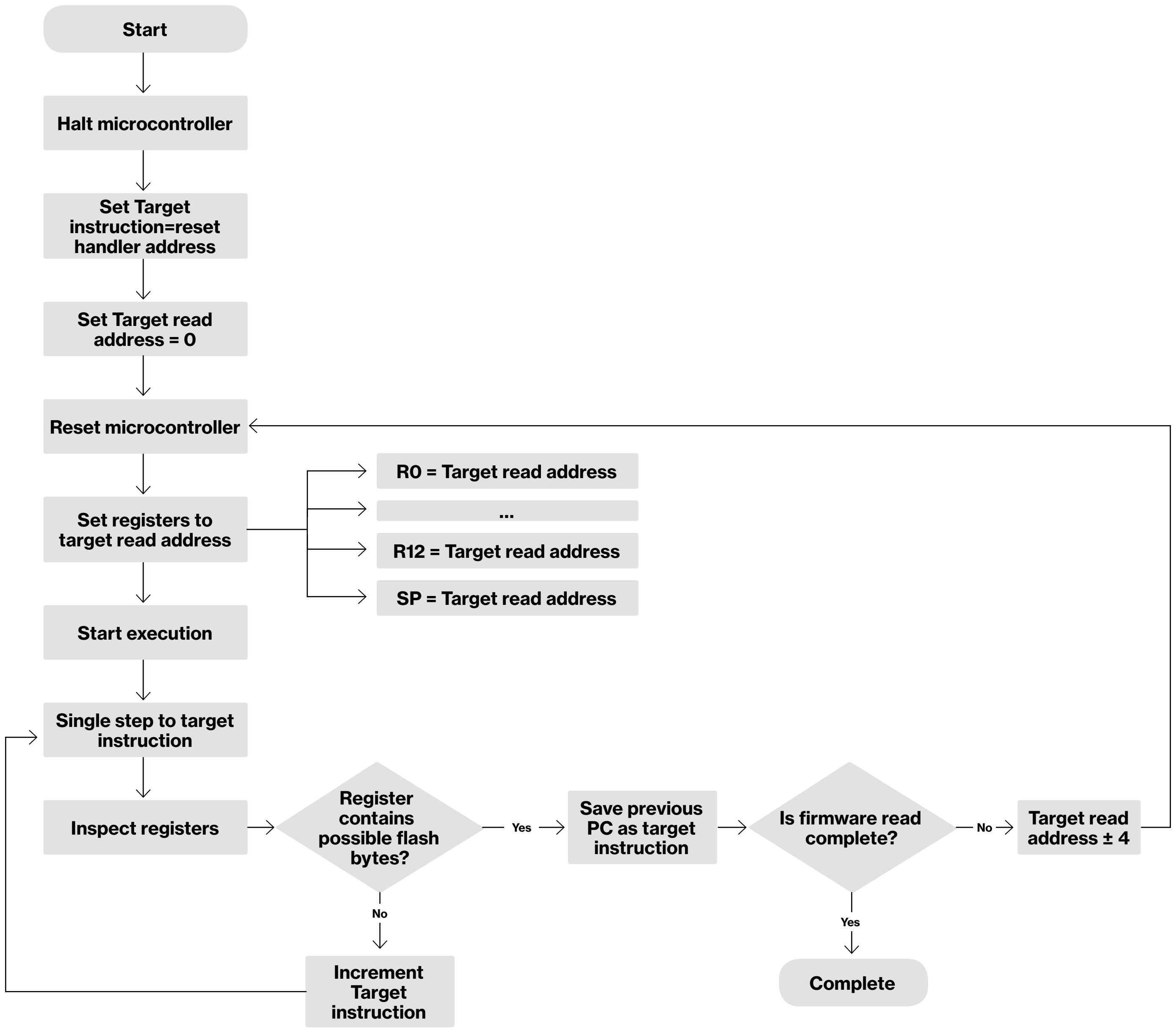
- **UICR.RBPCONF.PR0 == 0x00**
- **UICR.RBPCONF.PALL == 0xFF**

The documentation states that when PR0 equals 0x00, readback protection is enabled for code region 0. This affirmed my understanding that readback protection is in use on the Athena.

Defeating Nordic Semiconductor Readback Protection

With recovery of the nRF51822 firmware impeded due to readback protection, I set out find any prior research in defeating Nordic Semiconductor readback protections. I found a blog post[3] from 2015 written by Kris Brosch on the Include Security blog. The technique described in this blog post seemed like exactly what I needed to use.

The methodology begins with an attempt to locate an instruction in the firmware that takes one register as an address and reads the contents of that address into another register. The following flow diagram illustrates this effort:



Start
Halt microcontroller
Set Target instruction=reset handler address
Set Target read address = 0
Reset microcontroller
Set registers to target read address
R0 = Target read address
...
R12 = Target read address
SP = Target read address
Start execution
Single step to target instruction
Inspect registers
Register contains possible flash bytes?
Yes
No
Increment Target instruction
Save previous PC as target instruction
Is firmware read complete?
No
Yes
Target read address ± 4
Complete

Flow chart showing Include Security method for defeating readback protection, circa 2015

[3] http://blog.includesecurity.com/2015/11/NordicSemi-ARM-SoC-Firmware-dumping-technique.html

I attempted to reproduce this methodology to read the firmware from the Athena device. However, when I single stepped the microcontroller, I found behavior that differed from what was shown in the Include Security blog post. In particular, I found that single stepping appears to be impacted by the readback protection. Shown below is the result of single stepping the microcontroller after performing a reset:

```
J-Link>h
J-Link>r
...
J-Link>s
000006D0:  00 00              MOVS     R0, R0
J-Link>s
000006D2:  00 00              MOVS     R0, R0
J-Link>s
000006D4:  00 00              MOVS     R0, R0
```

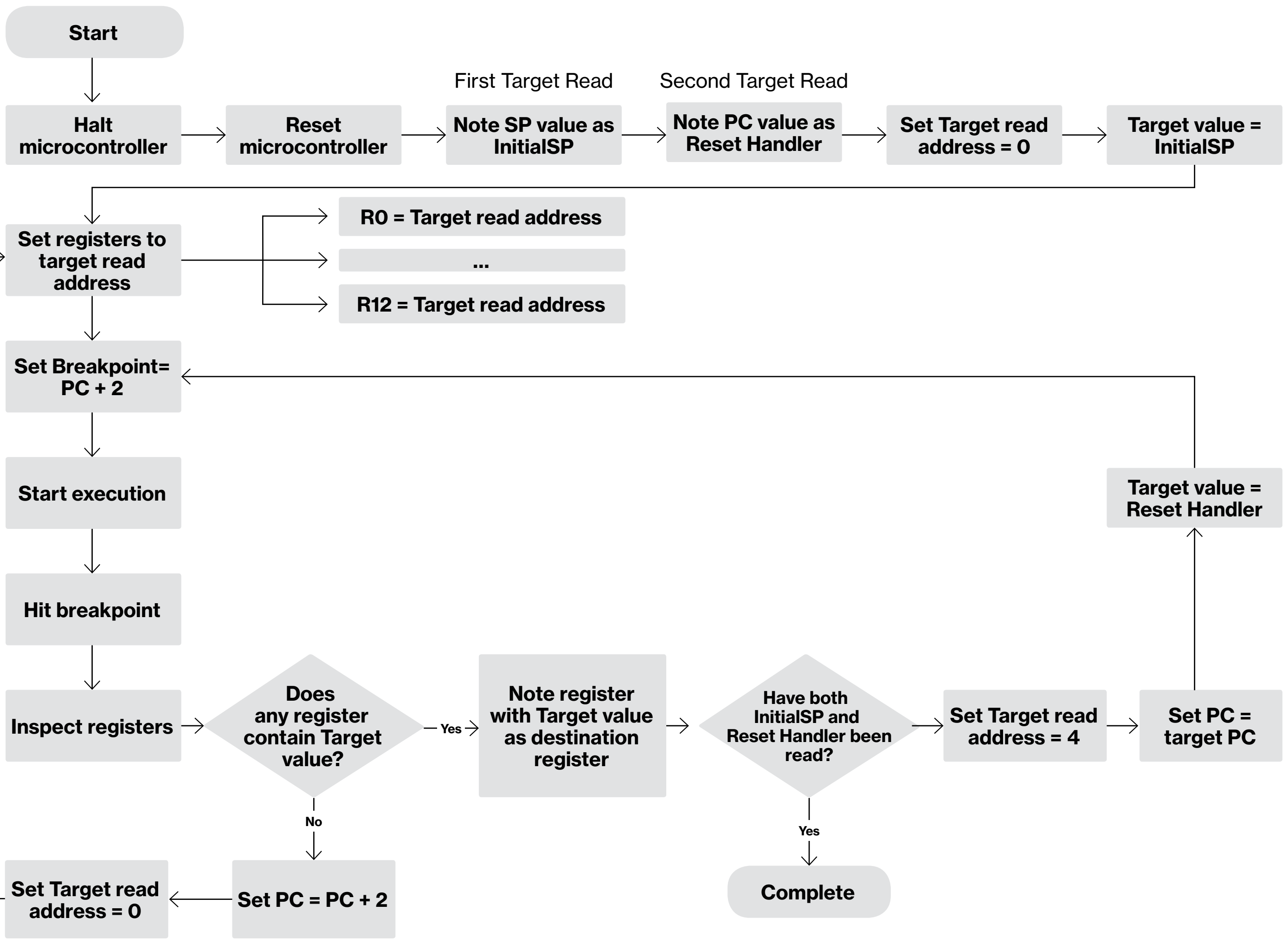
As shown, JLink Commander is single stepping through instructions comprised only of NULL bytes. My guess was that Nordic Semiconductor hardened the readback protection to prevent a debugger from retrieving the proper instruction bytes when single stepping.

After seeing this failure in my testing, I reconsidered the methodology based on my knowledge of ARM Cortex code layout. I arrived at a refined method for identifying the address of an appropriate instruction to read the contents of memory into a register. This method takes advantage of the way ARM Cortex microcontrollers behave when being reset by the debugger. The following output of JLink Commander shows the state of two key registers when resetting the microcontroller.

```
J-Link>r
J-Link>regs
PC = 000006D0, CycleCnt = 00000001
... REDACTED ...
SP(R13)= 000007C0, MSP= 000007C0, PSP= FFFFFFFC, R14(LR) = FFFFFFFF
```

The important registers to note are the PC register and the SP register. As mentioned previously, the first four bytes of an ARM Cortex firmware are the InitialSP value, and the second four bytes are the reset vector pointer. When the debugger puts the microcontroller into a reset state, the SP and PC values are set to the firmware’s InitialSP and reset vector, respectively. This debugger output shows the InitialSP value is 0x000007C0 and the reset vector value is 0x000006C0. We now have concrete values we can check against register values when single stepping the processor.

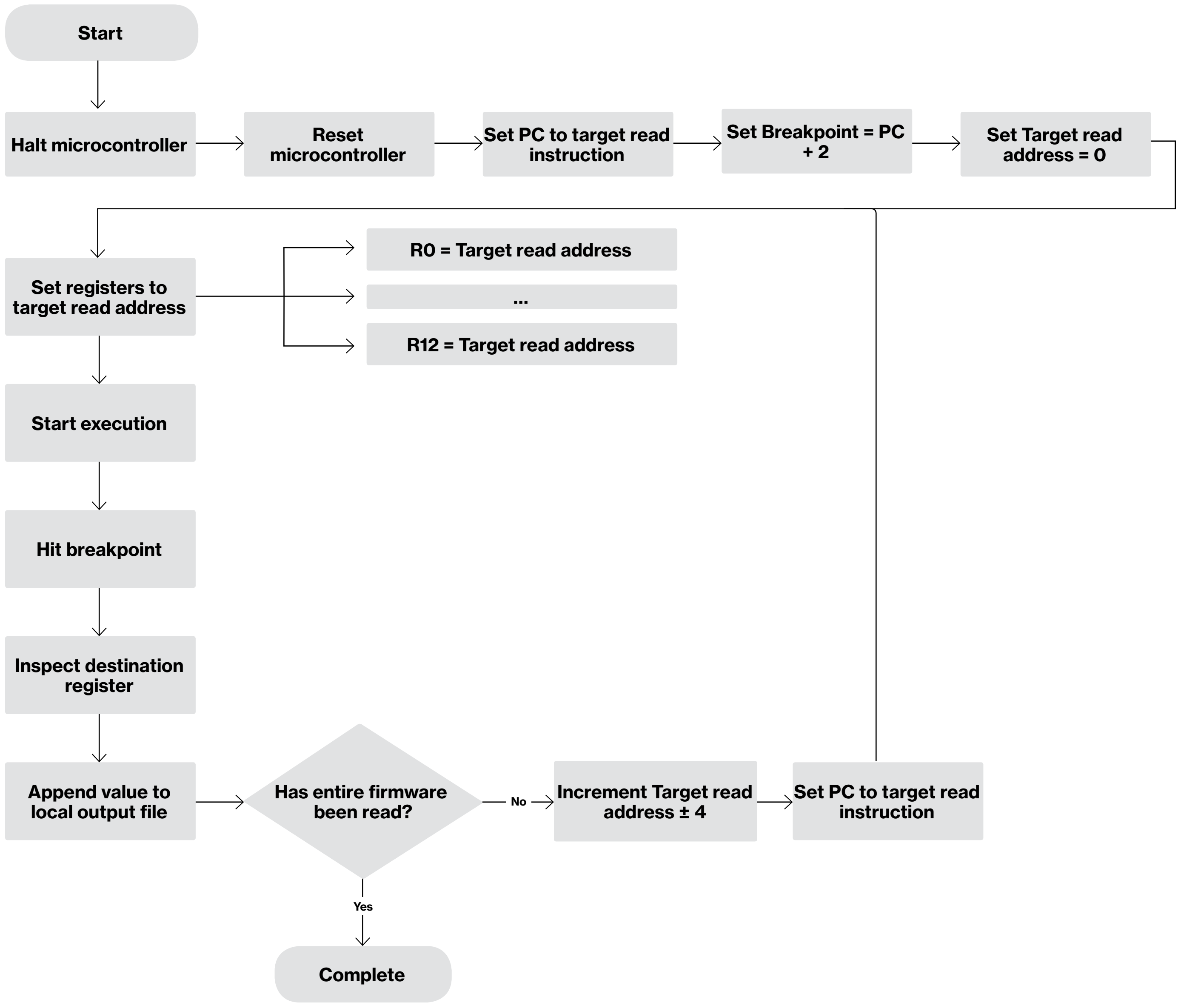
The methodology for finding a target instruction that reads four bytes from memory into a register can now be refined as shown in the following flow diagram:


Start
First Target Read
Second Target Read
Halt microcontroller
Reset microcontroller
Note SP value as InitialSP
Note PC value as Reset Handler
Set Target read address = 0
Target value = InitialSP
Set registers to target read address
R0 = Target read address
...
R12 = Target read address
Set Breakpoint= PC + 2
Start execution
Target value = Reset Handler
Hit breakpoint
Inspect registers
Does any register contain Target value?
Yes
Note register with Target value as destination register
Have both InitialSP and Reset Handler been read?
Set Target read address = 4
Set PC = target PC
No
Yes
Set Target read address = 0
Set PC = PC + 2
Complete

Flow chart showing updated method to discover useful instruction to defeat readback protection

Stated another way, since single stepping no longer functions the way we need it to, this method uses a breakpoint that causes the microcontroller to execute only one instruction. Also, because we know what value should be read into a register for the InitialSP, the guesswork is removed. When any register equals 0x000006D0, we know we have a likely target instruction address. Explicitly setting the PC register also makes extracting the entire firmware quicker, because each word being read out only needs to execute a single instruction instead of single stepping multiple instructions.

Once the target instruction is identified, extracting the firmware becomes straightforward:


Start
Halt microcontroller
Reset microcontroller
Set PC to target read instruction
Set Breakpoint = PC + 2
Set Target read address = 0
Set registers to target read address
R0 = Target read address
...
R12 = Target read address
Start execution
Hit breakpoint
Inspect destination register
Append value to local output file
Has entire firmware been read?
No
Increment Target read address ± 4
Set PC to target read instruction
Yes
Complete

Flow chart showing the firmware extraction process

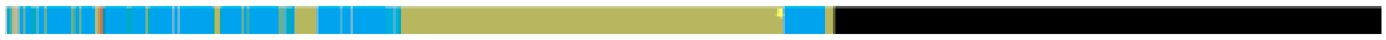
This breakpoint method gets around the new behavior of single stepping and allows the firmware to be completely read from the debugger. An attached debugger can still read the SP and PC registers, which also increases confidence when searching for a target instruction to use for bypassing readback protection. The first two values read in a successful bypass are the values populated into SP and PC when the device is reset.

Reading the entire 256k of flash is slow, but it works. Read speed can be increased by determining what instruction mnemonics are being used to read the contents of memory. Determining the register used as the source address, for example, limits the number of registers the JLink Commander script must update for every read.

Importing Athena Firmware into IDA Pro

The first thing to do after extracting the firmware is to apply the IDAPython code. The development time to produce a minimal but effective ARM Cortex M firmware loader pays dividends by identifying a relatively high amount of the code present in the firmware.

As with the Revolar Instinct firmware, loading the Athena firmware into IDA Pro is as simple as starting a new database, setting the architecture to ARM and setting the sub-architecture to ARMv7-M.



IDA Pro legend showing code identified in Athena firmware

As shown above, the IDAPython code has worked well in identifying quite a bit of code. These modules are discussed in greater detail in Appendix B. These modules are also available at Github.

3.0

Conclusion

In the course of this research, I spent quite a bit of time understanding more about ARM Cortex M microcontrollers and focusing on tools that helped me disassemble the firmware in IDA Pro. Performing this disassembly helped further my understanding of the firmware in both devices. While no vulnerabilities were discovered in the firmware, the project resulted in some useful tools for examining ARM microcontrollers. Others have found these tools useful for engaging in ARM Cortex disassembly. I expect to continue research into embedded ARM devices and will update the code repositories with more useful tools in the future.

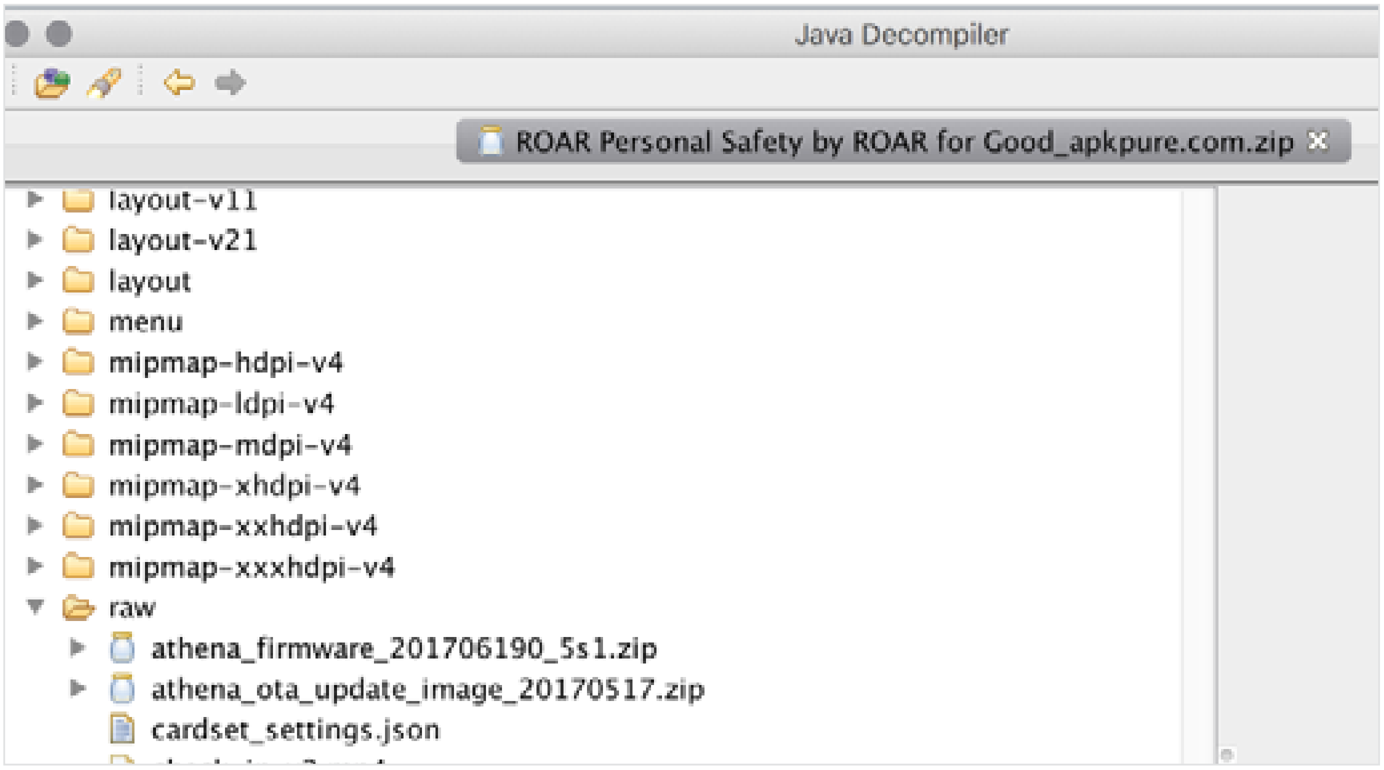
Appendices

Appendix A
Athena ROAR Android App

Over the course of this research project, I performed reverse engineering of Android applications for each personal protection device in an effort to better understand the Bluetooth Low Energy (BTLE) communications used between the mobile application and the personal protection devices. One particularly interesting discovery was the presence of partial firmware located in the application APK.

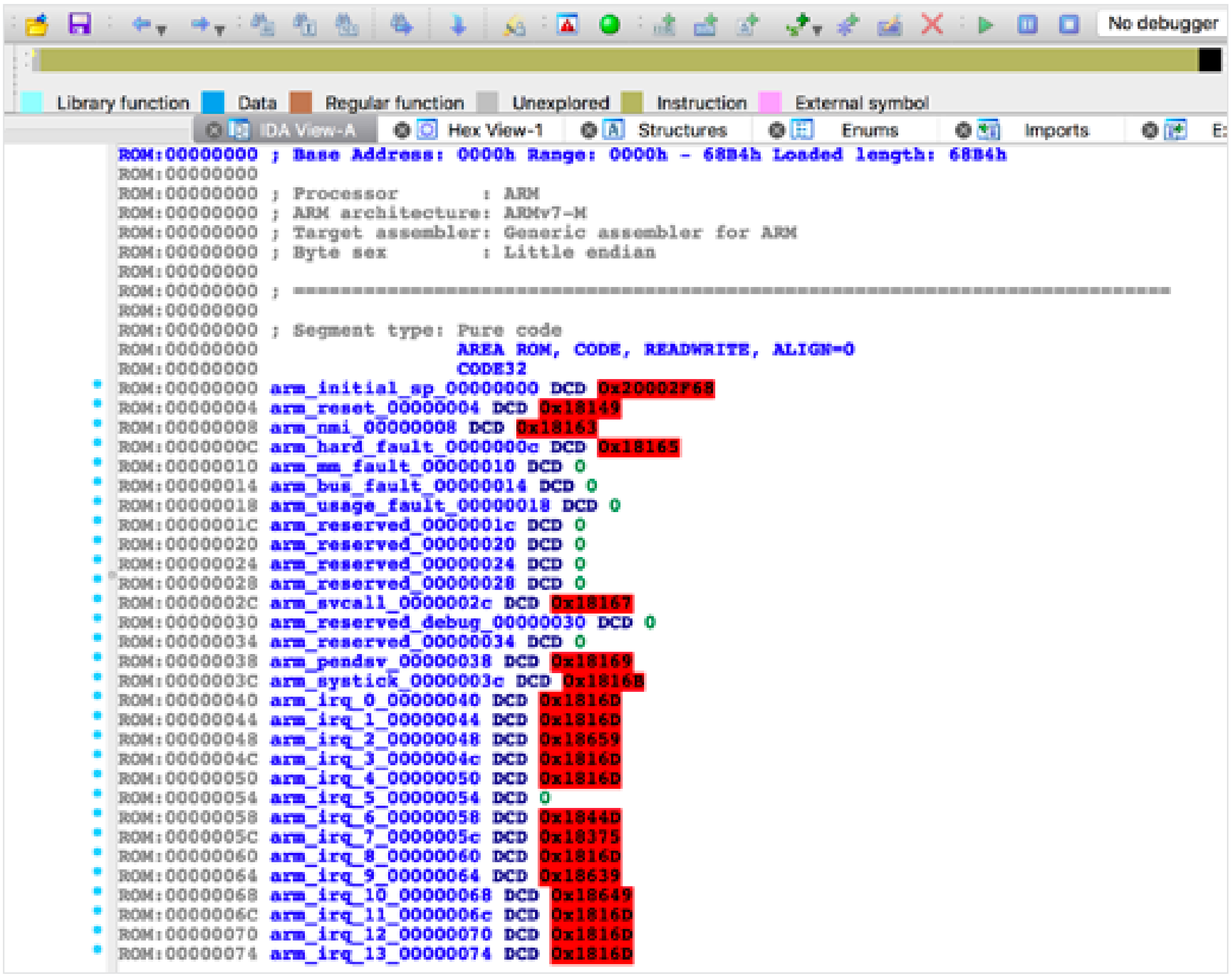
When I discovered readback protection in use on the Athena device, I wasn't sure if it was possible to defeat the protection. When I'm met with a problem, I will sometimes shelve the work that's being impeded and move on to some other aspect of the research. In this case, when I encountered the readback protection, I decided to pivot to performing analysis of the associated ROAR Personal Safety Android application.

Upon finding the firmware image contained in the Android application resources directory, I unzipped the firmware image. I loaded this firmware into IDA Pro for analysis, and noticed something strange about this firmware right away. None of the pointers in the vector table pointed at addresses inside the firmware image.



```
▶ layout-v11
▶ layout-v21
▶ layout
▶ menu
▶ mipmap-hdpi-v4
▶ mipmap-ldpi-v4
▶ mipmap-mdpi-v4
▶ mipmap-xhdpi-v4
▶ mipmap-xxhdpi-v4
▶ mipmap-xxxhdpi-v4
▼ raw
    ▶ athena_firmware_201706190_5s1.zip
    ▶ athena_ota_update_image_20170517.zip
      cardset_settings.json
```

JD GUI showing the presence of firmware in the Athena ROAR Android application



```
ROM:00000000 ; Base Address: 0000h Range: 0000h - 68B4h Loaded length: 68B4h
ROM:00000000
ROM:00000000 ; Processor       : ARM
ROM:00000000 ; ARM architecture: ARMv7-M
ROM:00000000 ; Target assembler: Generic assembler for ARM
ROM:00000000 ; Byte sex        : Little endian
ROM:00000000
ROM:00000000 ; ===========================================================================
ROM:00000000
ROM:00000000 ; Segment type: Pure code
ROM:00000000                 AREA ROM, CODE, READWRITE, ALIGN=0
ROM:00000000                 CODE32
ROM:00000000 arm_initial_sp_00000000 DCD 0x20002F68
ROM:00000004 arm_reset_00000004 DCD 0x18149
ROM:00000008 arm_nmi_00000008 DCD 0x18163
ROM:0000000C arm_hard_fault_0000000c DCD 0x18165
ROM:00000010 arm_mm_fault_00000010 DCD 0
ROM:00000014 arm_bus_fault_00000014 DCD 0
ROM:00000018 arm_usage_fault_00000018 DCD 0
ROM:0000001C arm_reserved_0000001c DCD 0
ROM:00000020 arm_reserved_00000020 DCD 0
ROM:00000024 arm_reserved_00000024 DCD 0
ROM:00000028 arm_reserved_00000028 DCD 0
ROM:0000002C arm_svcall_0000002c DCD 0x18167
ROM:00000030 arm_reserved_debug_00000030 DCD 0
ROM:00000034 arm_reserved_00000034 DCD 0
ROM:00000038 arm_pendsv_00000038 DCD 0x18169
ROM:0000003C arm_systick_0000003c DCD 0x1816B
ROM:00000040 arm_irq_0_00000040 DCD 0x1816D
ROM:00000044 arm_irq_1_00000044 DCD 0x1816D
ROM:00000048 arm_irq_2_00000048 DCD 0x18659
ROM:0000004C arm_irq_3_0000004c DCD 0x1816D
ROM:00000050 arm_irq_4_00000050 DCD 0x1816D
ROM:00000054 arm_irq_5_00000054 DCD 0
ROM:00000058 arm_irq_6_00000058 DCD 0x1844D
ROM:0000005C arm_irq_7_0000005c DCD 0x18375
ROM:00000060 arm_irq_8_00000060 DCD 0x1816D
ROM:00000064 arm_irq_9_00000064 DCD 0x18639
ROM:00000068 arm_irq_10_00000068 DCD 0x18649
ROM:0000006C arm_irq_11_0000006c DCD 0x1816D
ROM:00000070 arm_irq_12_00000070 DCD 0x1816D
ROM:00000074 arm_irq_13_00000074 DCD 0x1816D
```

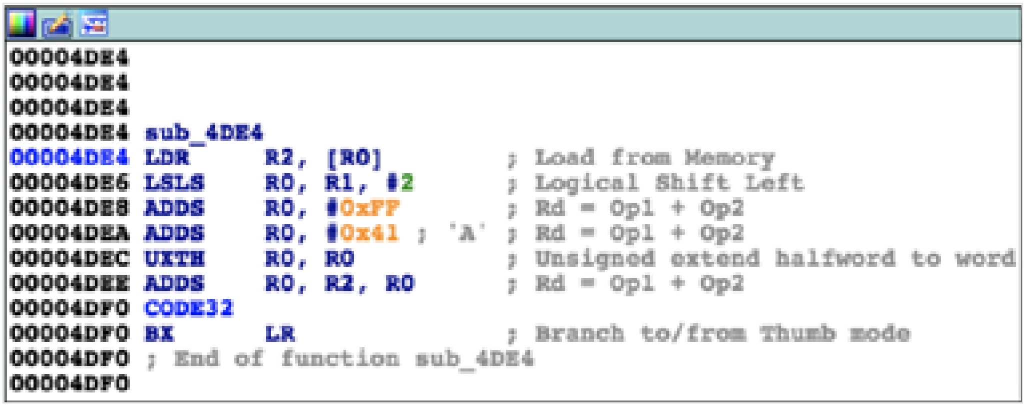
IDA Pro showing vector table. Red indicates addresses outside the range of addresses in the firmware

This result seemed odd. Typically ARM Cortex firmware vectors point to actual functions. This result was hinting at something that now seems obvious; this firmware image does not begin at address 0x0 on the device. In other words, in IDA Pro, this firmware needed to be rebased to start at a different address.

This didn't occur to me at the time. I found these pointers to be a curiosity, but did not dig into why they were apparently wrong. I wanted to tackle the readback protection problem.

When I first found this firmware, I thought I could ignore the readback protection. I presumed this firmware image was the complete firmware running in the Athena device. That's a pretty big assumption with no supporting data. So, while I disassembled the firmware recovered from the Android application, I wanted to verify the contents of the device flash in order to determine whether the firmware recovered from the Android app was indeed identical to the device firmware.

As noted earlier, prior research on defeating Nordic Semiconductor readback protection described the use of reset handler instructions to iteratively copy bytes from memory into a register. Instead of using the reset handler, I decided to use an instruction from within this firmware image.



```
00004DE4
00004DE4
00004DE4
00004DE4 sub_4DE4
00004DE4 LDR     R2, [R0]          ; Load from Memory
00004DE6 LSLS    R0, R1, #2        ; Logical Shift Left
00004DE8 ADDS    R0, #0xFF         ; Rd = Op1 + Op2
00004DEA ADDS    R0, #0x41 ; 'A'   ; Rd = Op1 + Op2
00004DEC UXTH    R0, R0            ; Unsigned extend halfword to word
00004DEE ADDS    R0, R2, R0        ; Rd = Op1 + Op2
00004DF0 CODE32
00004DF0 BX      LR                ; Branch to/from Thumb mode
00004DF0 ; End of function sub_4DE4
00004DF0
```

IDA Pro showing instruction “LDR R2, [R0]” in the recovered firmware

I chose this instruction, located at address 0x4DE4, as the memory read instruction I would attempt to use to read the full 256k flash from the Nordic Semiconductor chip. I chose this instruction because R0 is treated as a pointer to memory and R2 will contain the contents of memory after that instruction is executed.

When setting the breakpoint just after the instruction above and single stepping, I expected the first four bytes of the firmware read into R2. This would have been the device’s InitialSP value. However, my breakpoint was never reached.

```
J-Link>regs
CPU is not halted !
J-Link>h
PC = 00018164, CycleCnt = 00000000
R0 = FFFFFFFF, R1 = FFFFFFFF, R2 = DEADBEEF, R3 = 00018165
R4 = FFFFFFFF, R5 = FFFFFFFF, R6 = FFFFFFFF, R7 = FFFFFFFF
R8 = 00000000, R9 = 00000000, R10= 00000000, R11= 00000000
R12= 00000000
SP(R13)= 000007E8, MSP= 000007E8, PSP= FFFFFFFC, R14(LR) = FFFFFFF9
```

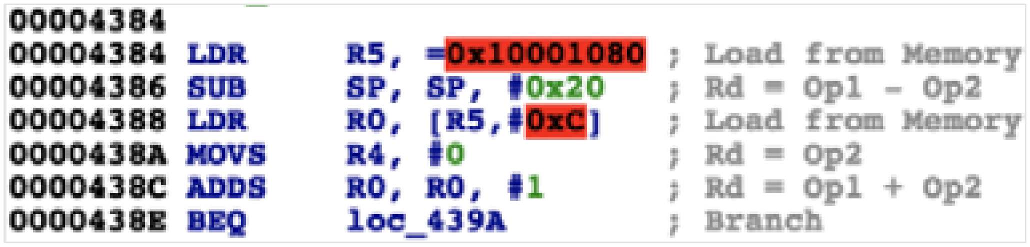
This excerpt from JLink Commander shows the instruction pointer (PC) is nowhere near my expected value of 0x4DE6. It’s off somewhere else. Also, most of the registers contain 0xFFFFFFFF and R2=0xDEADBEEF. This looked suspicious.

I realized that I needed to rebase the firmware in IDA Pro. With the luxury of hindsight, the rebase offset seems obvious. At the time, however, the answer was not so clear. I decided to use some dynamic monitoring of BTLE communications in combination with the debugger access.

I had noted the firmware defined a number of strings, several of which corresponded to messages being transmitted from the device over BTLE. My strategy was to monitor BTLE traffic and halt the device when I saw one of the known strings transmitted. Examples of data I would be looking for include the manufacturer, model number and serial number. When I halted the microcontroller, I saw the following details in JLink Commander:

```
J-Link>h
PC = 0000D16C, CycleCnt = 00000000
R0 = 2000007F, R1 = 00000007, R2 = E000E200, R3 = 00000007
R4 = 00000000, R5 = 10001080, R6 = 0001E884, R7 = 00003000
R8 = FFFFFFFF, R9 = FFFFFFFF, R10= FFFFFFFF, R11= FFFFFFFF
R12= FFFFFFFF
SP(R13)= 20002F30, MSP= 20002F30, PSP= FFFFFFFC, R14(LR) = 0000119B
```

I searched the firmware image for any instruction that set the registers as shown above. Good fortune smiled on me that day, because I found only a single instruction in the firmware that set the R5 register.



```
00004384
00004384 LDR     R5, =0x10001080    ; Load from Memory
00004386 SUB     SP, SP, #0x20      ; Rd = Op1 - Op2
00004388 LDR     R0, [R5,#0xC]      ; Load from Memory
0000438A MOVS    R4, #0             ; Rd = Op2
0000438C ADDS    R0, R0, #1         ; Rd = Op1 + Op2
0000438E BEQ     loc_439A           ; Branch
```

IDA Pro showing an instruction setting R5=0x10001080. The only such instruction in the firmware.

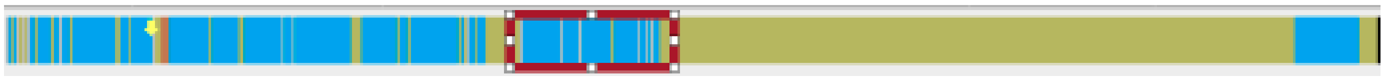
Working off the assumption that this instruction exists in the device firmware, I set a watchpoint in my debugger on address 0x1000108C in order to halt the microcontroller whenever an access is made at that memory location. In the above image, the instruction at address 0x4388 will trigger the watchpoint. This is because the target address of 0x10001080 is loaded into R5 at the beginning of the function and then dereferenced in the third instruction.

```
J-Link>regs
PC = 0001C38A, CycleCnt = 00000000
R0 = 30315252, R1 = 20002F68, R2 = 00000000, R3 = 00018787
R4 = 0001E884, R5 = 10001080, R6 = 0001E884, R7 = 00003000
R8 = FFFFFFFF, R9 = FFFFFFFF, R10= FFFFFFFF, R11= FFFFFFFF
R12= FFFFFFFF
```

By observing the value of the PC register, I saw the instruction pointer point to the instruction directly after the expected watchpoint was hit. Doing a bit of math (0x1C38A - 0x438A) shows the firmware recovered from the Android app should be rebased to address 0x18000. Once I rebase the firmware in IDA Pro, the reset vectors all point to valid code.

The result of having a properly rebased firmware image in IDA Pro is that now I can attempt to perform the readback protection bypass using the instruction I identified earlier (LDR R2, [R0]). I used the modified readback protection bypass to iteratively hit a breakpoint at address 0x1CDE4 and read out the contents of the R2 register.

This firmware continued with surprises. I found that the entire firmware recovered from the Android application resides in the device flash located at offset 0x18000 just as expected.



IDA Pro showing the entire Athena firmware. The red rectangle indicates the firmware recovered from the ROAR Android application.

This result was interesting - while the Android application did afford us a partial firmware image, if the readback protection bypass had not worked, a large part of the firmware would not have been available for analysis.

Appendix B
IDAPython for ARM Cortex Firmware Analysis

During the course of this research, I developed some IDAPython code to load ARM Cortex firmware images in IDAPython. Currently there are two modules: cortex_m_firmware.py and amnesia.py. The Cortex M module can annotate Cortex vector tables in the firmware. The Amnesia module implements the heuristics for ARM code and function discovery.

This code is available **here** on the Duo Labs github.

In addition to the heuristics that find functions based on byte patterns in the IDA database, I also developed heuristic functions that attempt to find function boundaries in assembly mnemonics. In some cases, converting sections of bytes into code using IDA Pro's APIs will not result in properly defined functions based on that code. The main technique used in this opcode-level heuristic is to search for sequences of code that are not defined in a function. The heuristic then converts the code to a new function if the first instruction mnemonic matches a list of predefined mnemonics: PUSH, STM or MOV. I selected these mnemonics due to their common use in function prologues. This technique is similar to the mnemonics mentioned above that operate at the byte level.

As I continue research into ARM Cortex microcontrollers, these IDAPython modules will continue to improve. I will develop additional code detection heuristics based on some of the manual reverse engineering techniques used during this research. I will also look at the function prologues generated by a number of different compilers in order to develop a more complete set of code-detection heuristics. I am also considering porting the IDAPython modules to other reverse engineering tools, such as Capstone, Hopper and Binary Ninja.

The current version of the tools we have released for IDA Pro must be run at the IDA Pro command line. The Github repository contains a README to show how to use the tools. The goal of this project was to develop easy-to-use IDAPython modules. Once you check out the IDAPython code from the Github repository, you must reconfigure IDAPython to include the path to the location to which you checked out the modules.

The Cortex M module wraps the Amnesia module with the goal of grooming Cortex M firmware as simply as possible. From the IDA Pro command line, the most straightforward way to import standard firmware is to simply instantiate the CortexMFirmware class and set auto to "True."

```
from cortex_m_firmware import *
cortex = CortexMFirmware(auto=True)
```

In this example, the code will annotate the vector table located at address 0x0, and then perform the heuristic code detection described above. If you have a firmware image that does not begin at address 0x0 in your database due to relocation, you need to specify the proper address of the vector table. The following excerpt also shows how to use the Cortex M module to clean up a firmware image that contains multiple vector tables.

```
from cortex_m_firmware import *
cortex = CortexMFirmware()
cortex.annotate_vector_table(0x4000)
cortex.annotate_vector_table(0x10000)
cortex.find_functions()
```

The code example above annotates two vector tables in the firmware, then calls the heuristic code-finding methods from the Amnesia module. The reverse engineer using this should inspect unconverted regions, as the current versions of these tools do not typically find all code in the firmware.

The IDAPython code has been written in a modular way to make it possible to use the Amnesia IDAPython module to find functions in an ARM binary without using the Cortex M module. Simply instantiate the Amnesia class, and call any of the functions in the module. The following shows the functions available in the Amnesia class.

```
class Amnesia:
 def find_function_epilogue_bxlr(self, makecode=False)
 def find_pushpop_registers_thumb(self, makecode=False)
 def find_pushpop_registers_arm(self, makecode=False)
 def make_new_functions_heuristic_push_regs(self, makefunction=False)
 def nonfunction_first_instruction_heuristic(self, makefunction=False)
```

Check out our Github repository and try this code against your ARM Cortex targets. Pull requests are welcome. If you want to talk ARM detection heuristics, I welcome your feedback as well.

Appendix C References

Revolar Instinct. https://revolar.com/

Athena. https://roarforgood.com/

EFR32BG1 Blue Gecko Bluetooth® Smart SoC Family Data Sheet. https://www.silabs.com/documents/login/data-sheets/efr32bg1-datasheet.pdf

Saleae Logic Analyzer. https://www.saleae.com/

Segger JLink debugging interface. https://www.segger.com/products/debug-probes/j-link/

Segger JLink Commander. https://www.segger.com/products/debug-probes/j-link/tools/j-link-commander/

GNU GDB debugger. https://www.gnu.org/software/gdb/

IDA Pro disassembler. https://www.hex-rays.com/products/ida/

JD GUI Java decompiler. http://jd.benow.ca/

About the Author

Todd Manning

Our mission is to protect your mission.

Experience advanced two-factor authentication, endpoint visibility, custom user policies & more with your free 30 day trial.

Duo Security makes security painless, so you can focus on what's important. Our scalable, cloud-based Trusted Access platform addresses security threats before they become a problem, by verifying the identity of your users and the health of their devices before they connect to the applications you want them to access.

Thousands of organizations worldwide use Duo, including Facebook, Toyota, Panasonic and MIT. Duo is backed by Google Ventures, True Ventures, Radar Partners, Redpoint Ventures and Benchmark. We're located from coast to coast and across the sea.

Try it today at duo.com.

Follow @duosec and @duo_labs on Twitter.

The Trusted Access Company

duo.com