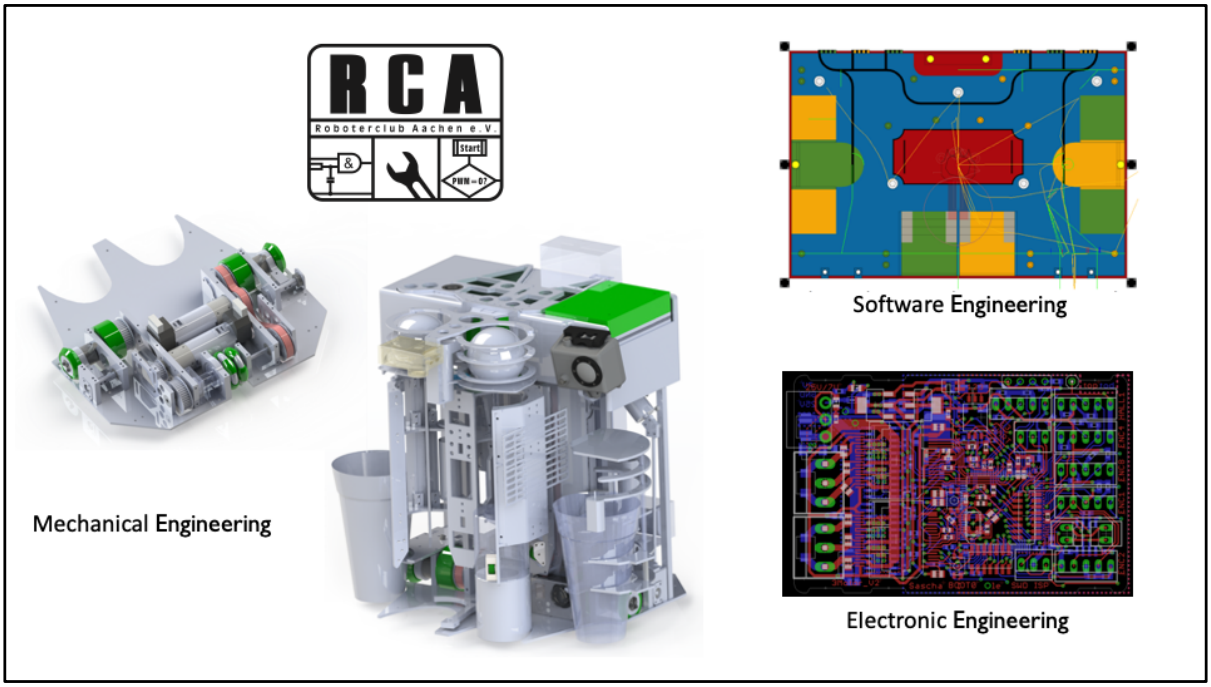
Introduction to ARMv8-M and TrustZone-M

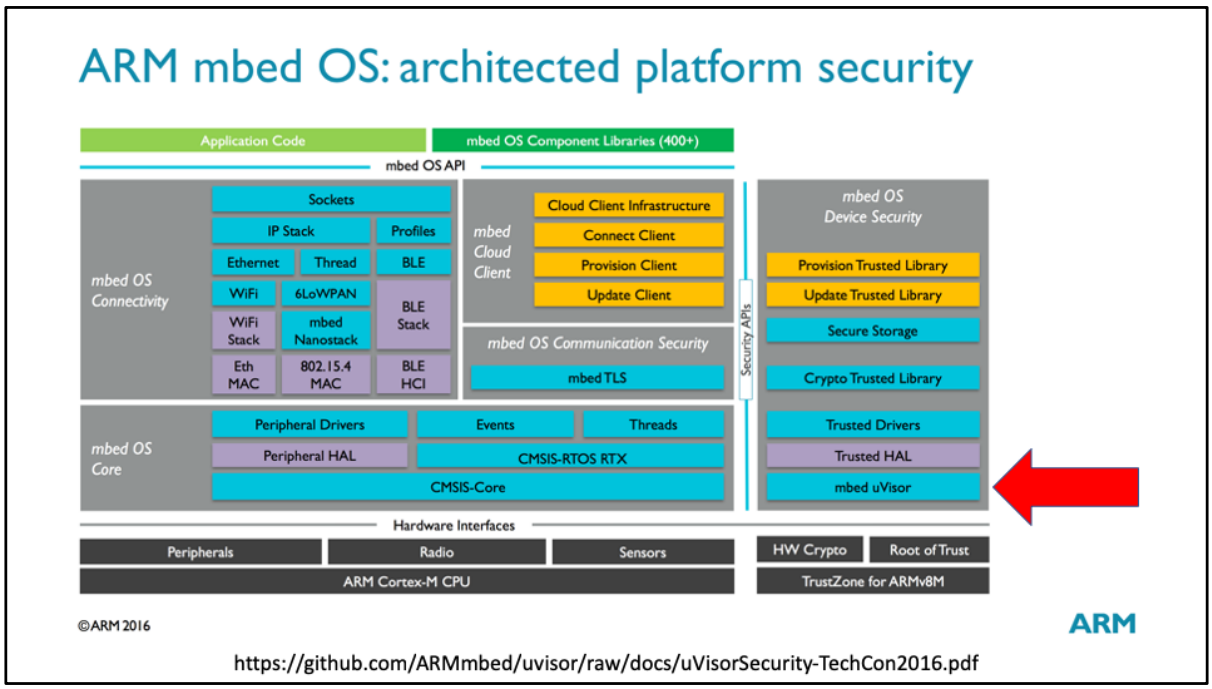
Niklas Hauser

emBO++ 2018

According to my parents I study something with computers.



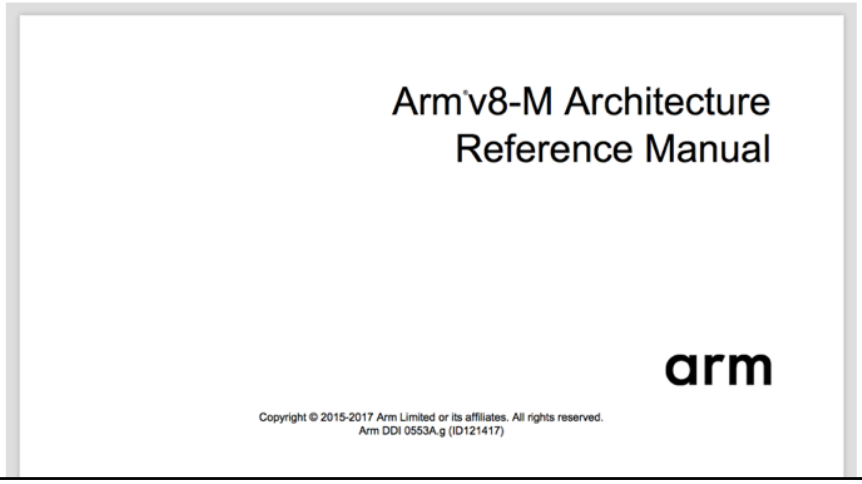
But I also build autonomous robots in my spare time.



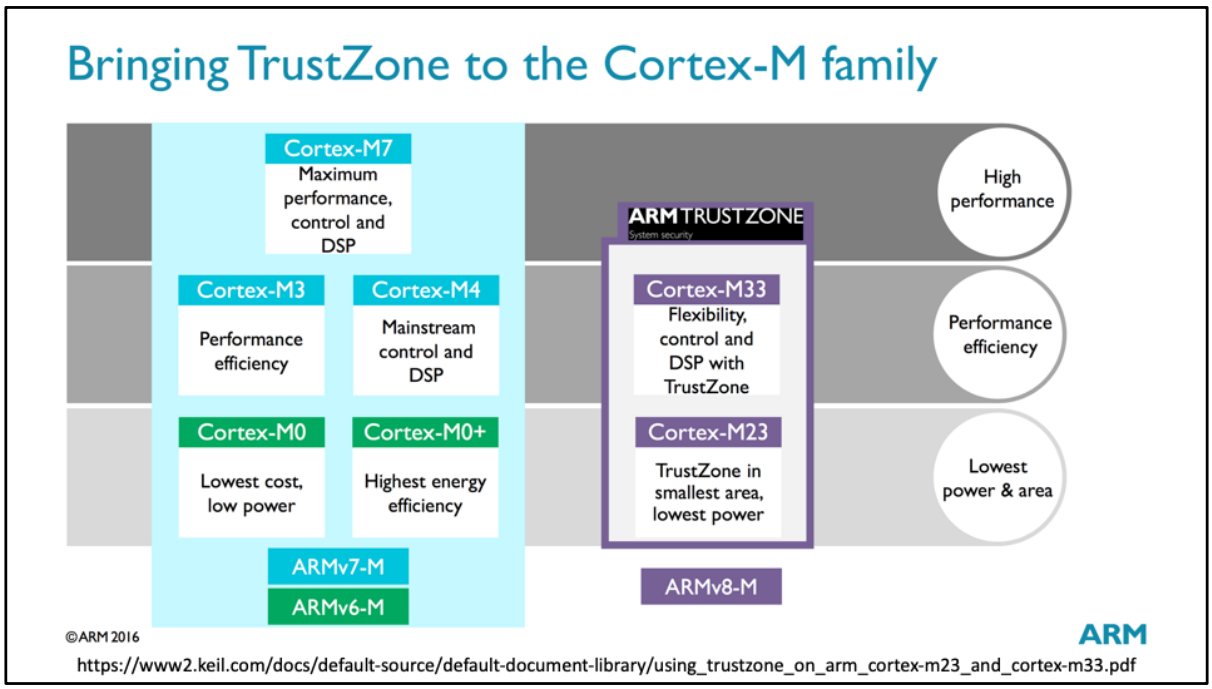
2.5y ago, I got hired by ARM to work on mbed OS Security, specifically uVisor.
I spent almost 2y on the lowest levels of the ARMv7-M and ARMv8-M architectures.
I've seen ALL the dirt of Cortex-M and I still (mostly) love it.

Then I quit to work on the largest research model railway in Germany.
YOLO.

Hello. My name is Niklas. And I would like to share with you the most amazing book!



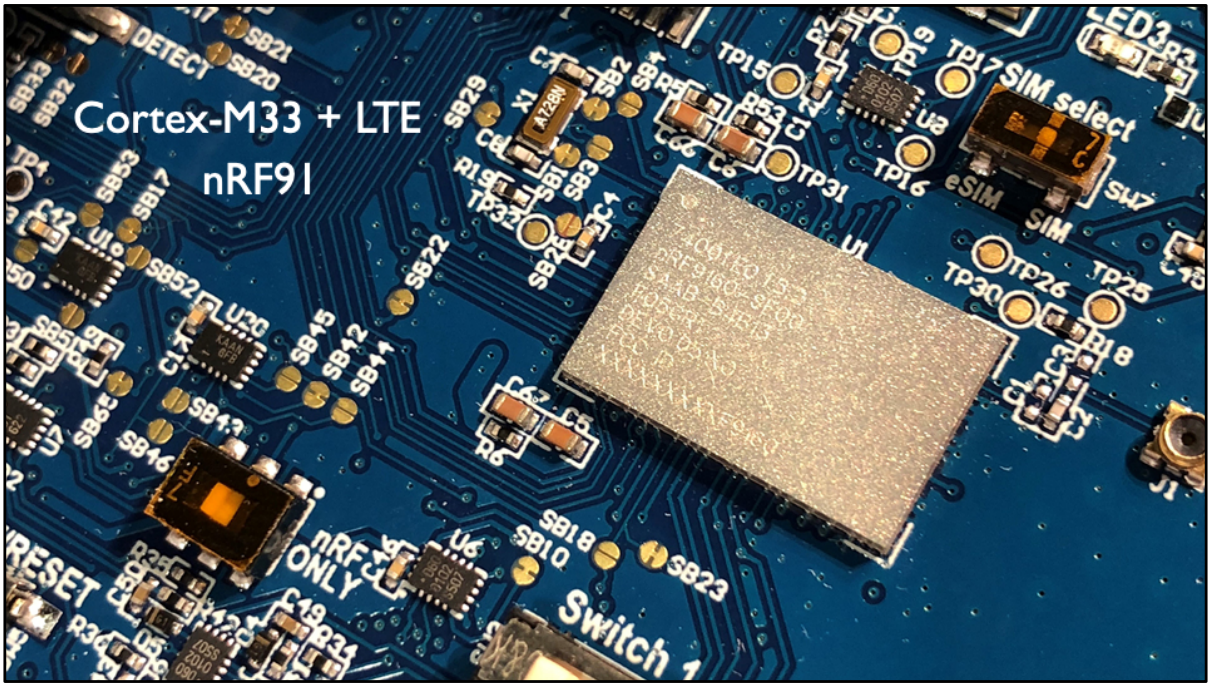
Music from Book of Mormon.



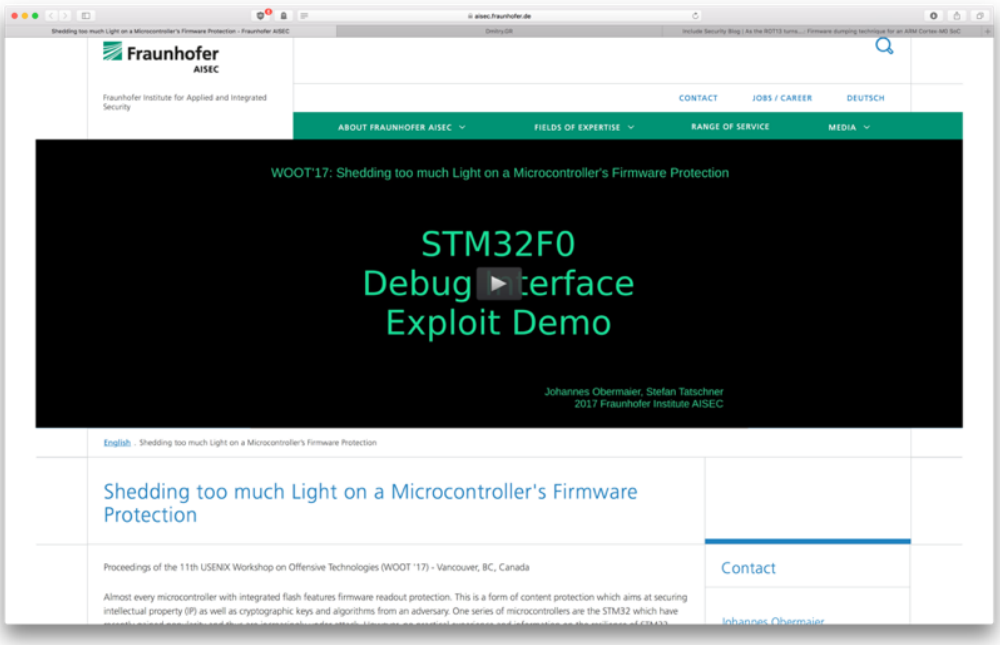
On the left are the current Cortex-Ms that you probably al know.
M0/M0+ for low power and cost
M3/M4 for performance
M7 for ludacris performance

And then in late 2015 ARM announced the ARMv8-M architecture.
Cortex-M23 was released on June 23rd 2016, aka the day of the Brexit vote!

Maybe M43 in future?



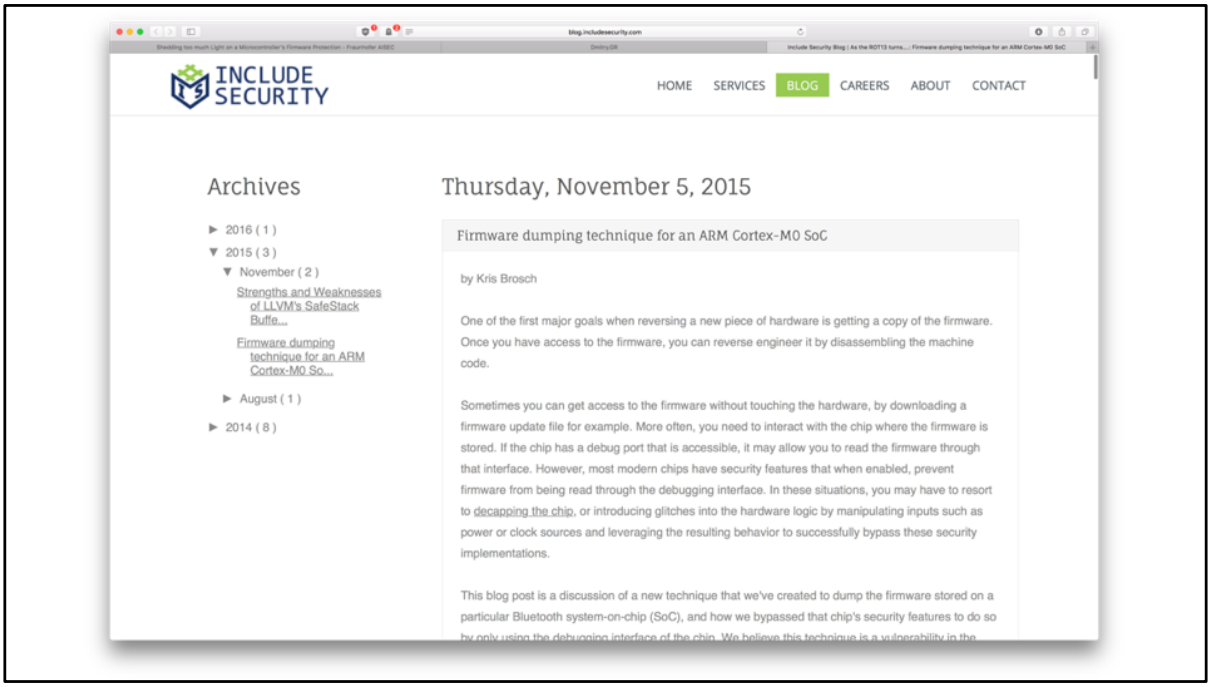
First ARMv8-M implementation shipped by Nordic: nRF91
Cellular Modem + CryptoCell + Cortex-M33



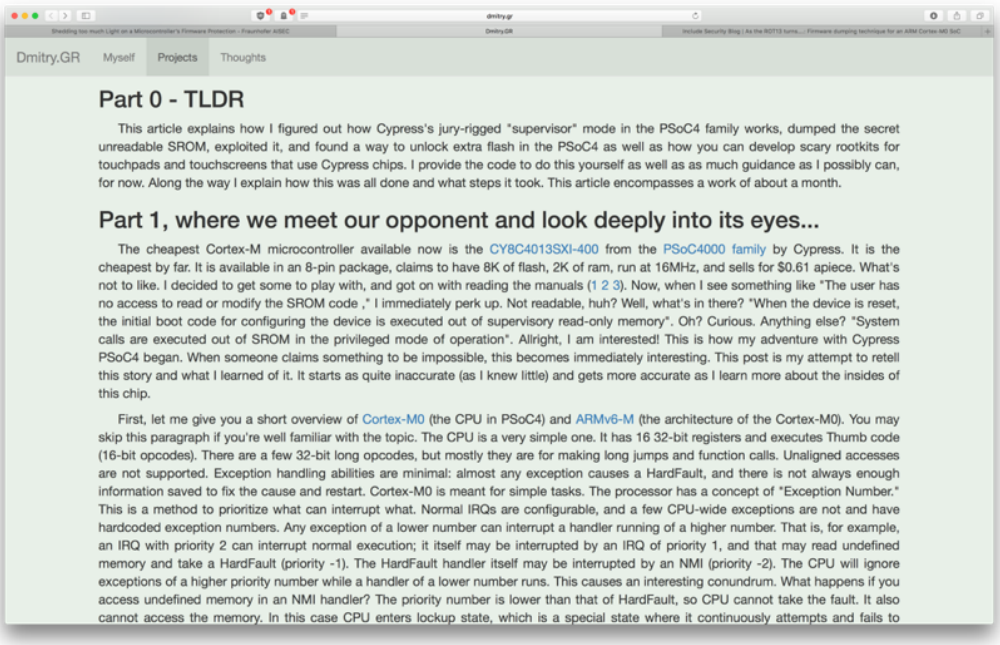
ARM only provides the CPU implementation, it DOES NOT provide any memory implementations.
So vendors have to integrate the CPU with Flash and RAM themselves.

⇒ Flash readout protection mechanisms are not unified
⇒ Recently a race condition in the debug interface allowed circumventing STM32 flash readout protection entirely!

Marc talked about this in his talk



NRF51 had too permissive debug permissions, allowed reading and modifying the CPU registers even when Flash readout protection was enabled.
With a bit of luck, entire flash could be dumped.



Dmitry.GR Myself Projects Thoughts

Part 0 - TLDR

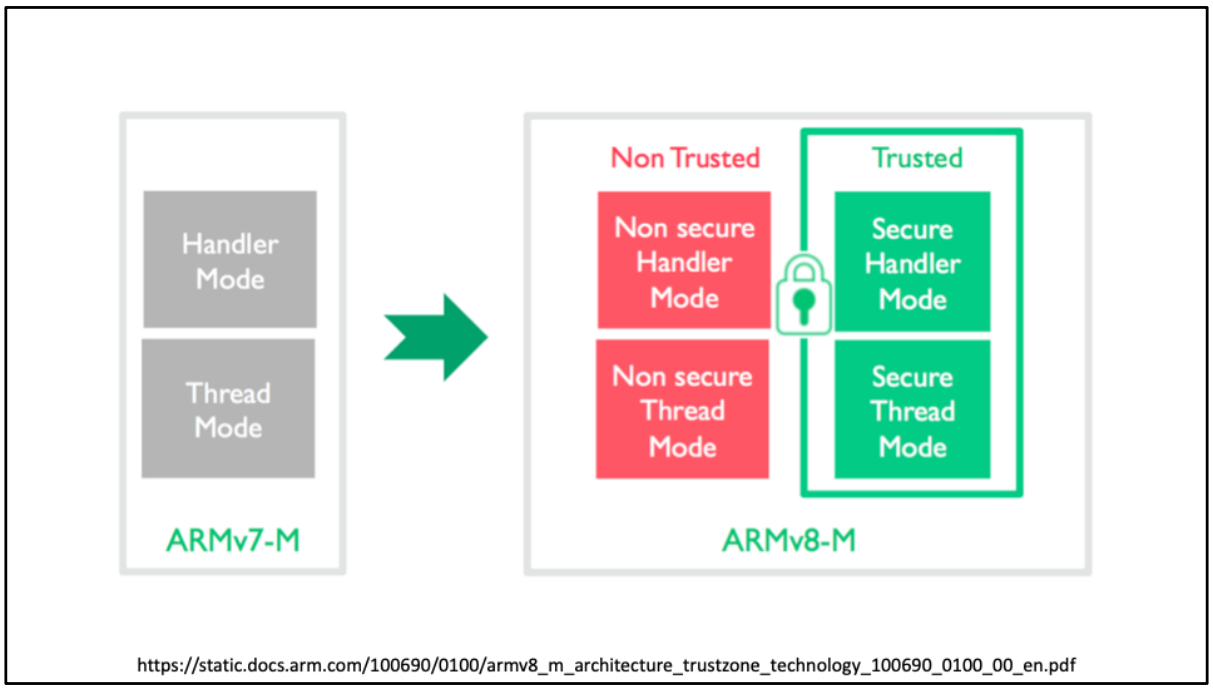
This article explains how I figured out how Cypress's jury-rigged "supervisor" mode in the PSoC4 family works, dumped the secret unreadable SROM, exploited it, and found a way to unlock extra flash in the PSoC4 as well as how you can develop scary rootkits for touchpads and touchscreens that use Cypress chips. I provide the code to do this yourself as well as as much guidance as I possibly can, for now. Along the way I explain how this was all done and what steps it took. This article encompasses a work of about a month.

Part 1, where we meet our opponent and look deeply into its eyes...

The cheapest Cortex-M microcontroller available now is the CY8C4013SXI-400 from the PSoC4000 family by Cypress. It is the cheapest by far. It is available in an 8-pin package, claims to have 8K of flash, 2K of ram, run at 16MHz, and sells for $0.61 apiece. What's not to like. I decided to get some to play with, and got on with reading the manuals (1 2 3). Now, when I see something like "The user has no access to read or modify the SROM code ," I immediately perk up. Not readable, huh? Well, what's in there? "When the device is reset, the initial boot code for configuring the device is executed out of supervisory read-only memory". Oh? Curious. Anything else? "System calls are executed out of SROM in the privileged mode of operation". Allright, I am interested! This is how my adventure with Cypress PSoC4 began. When someone claims something to be impossible, this becomes immediately interesting. This post is my attempt to retell this story and what I learned of it. It starts as quite inaccurate (as I knew little) and gets more accurate as I learn more about the insides of this chip.

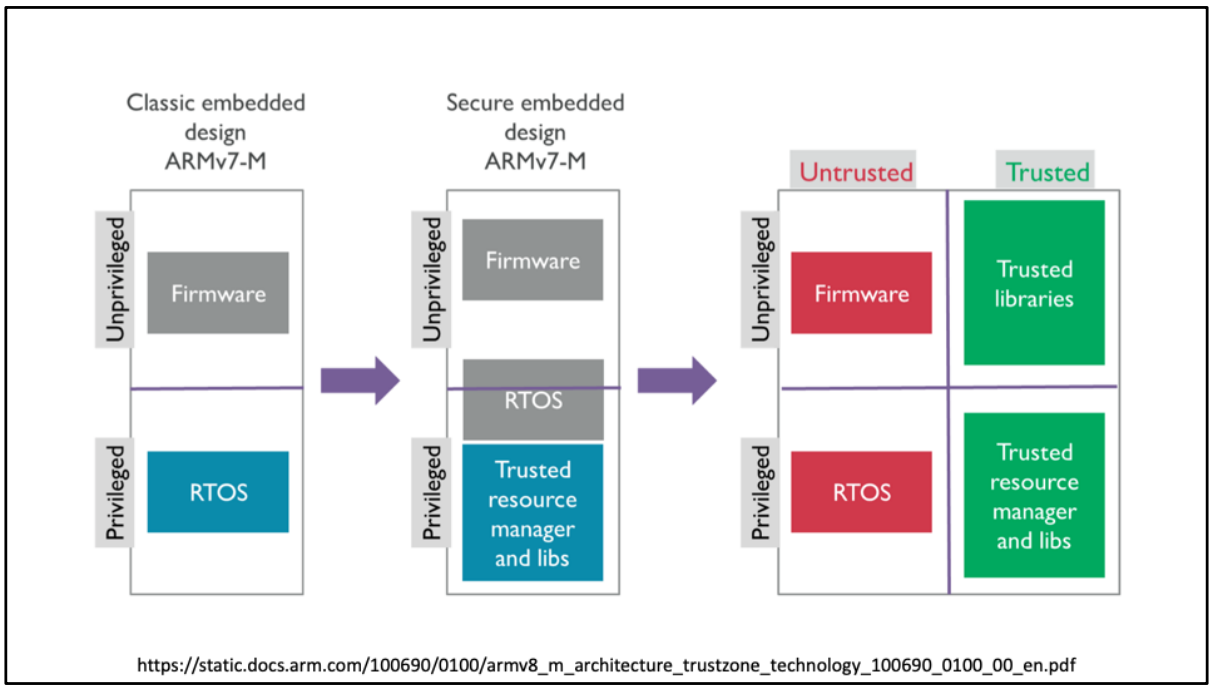
First, let me give you a short overview of Cortex-M0 (the CPU in PSoC4) and ARMv6-M (the architecture of the Cortex-M0). You may skip this paragraph if you're well familiar with the topic. The CPU is a very simple one. It has 16 32-bit registers and executes Thumb code (16-bit opcodes). There are a few 32-bit long opcodes, but mostly they are for making long jumps and function calls. Unaligned accesses are not supported. Exception handling abilities are minimal: almost any exception causes a HardFault, and there is not always enough information saved to fix the cause and restart. Cortex-M0 is meant for simple tasks. The processor has a concept of "Exception Number." This is a method to prioritize what can interrupt what. Normal IRQs are configurable, and a few CPU-wide exceptions are not and have hardcoded exception numbers. Any exception of a lower number can interrupt a handler running of a higher number. That is, for example, an IRQ with priority 2 can interrupt normal execution; it itself may be interrupted by an IRQ of priority 1, and that may read undefined memory and take a HardFault (priority -1). The HardFault handler itself may be interrupted by an NMI (priority -2). The CPU will ignore exceptions of a higher priority number while a handler of a lower number runs. This causes an interesting conundrum. What happens if you access undefined memory in an NMI handler? The priority number is lower than that of HardFault, so CPU cannot take the fault. It also cannot access the memory. In this case CPU enters lockup state, which is a special state where it continuously attempts and fails to

PSoC4 uses a hardware+software solultion to restrict access to certain memories in software! Ships 16kB RAM, enables only 8kB via supervisor task running in privileged mode.
Broken to unlock full RAM and more.



What do you do to make things more secure? SLAP A GREEN LOCK ON IN. like web browsers.
Green lock means Trusted. BAM, problem solved, thanks, goodbye.

Important to place the untrusted side on the LEFT and make it RED.
Like some kind of data communists, that want to share all the data fairly.

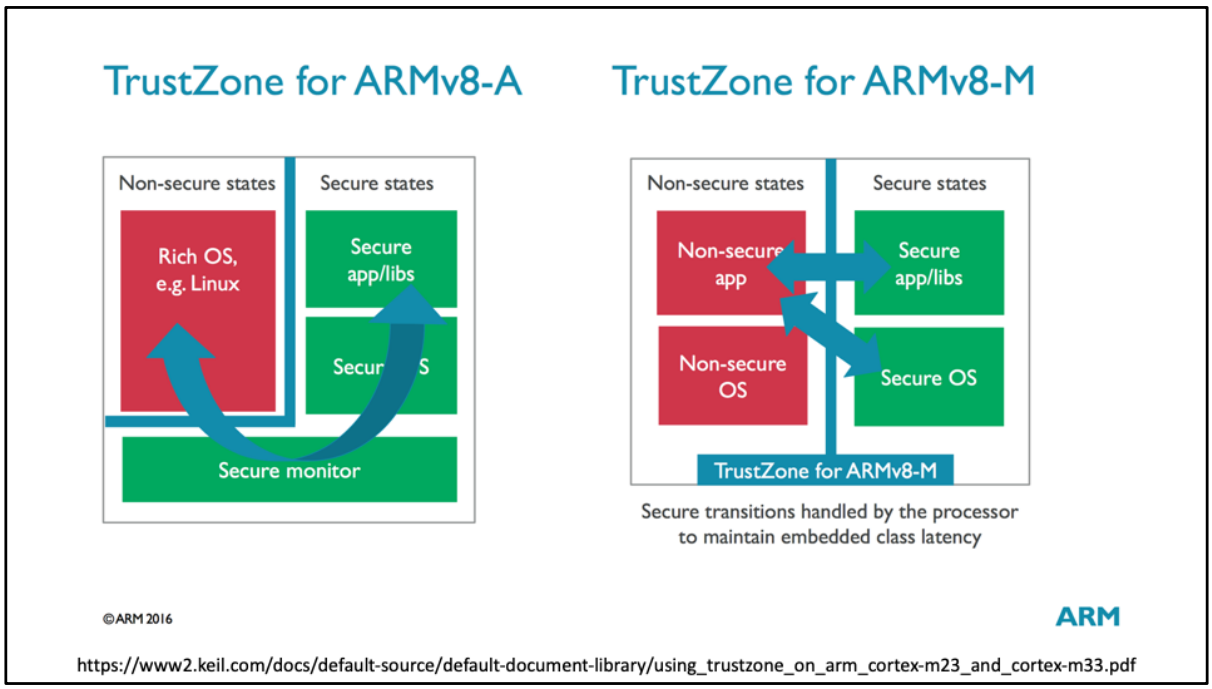


A bit more of a technical slide:
There are two mode “dimentions”:

1. (un-) privileged as a Safety domain
2. (un-) trusted as a Security domain

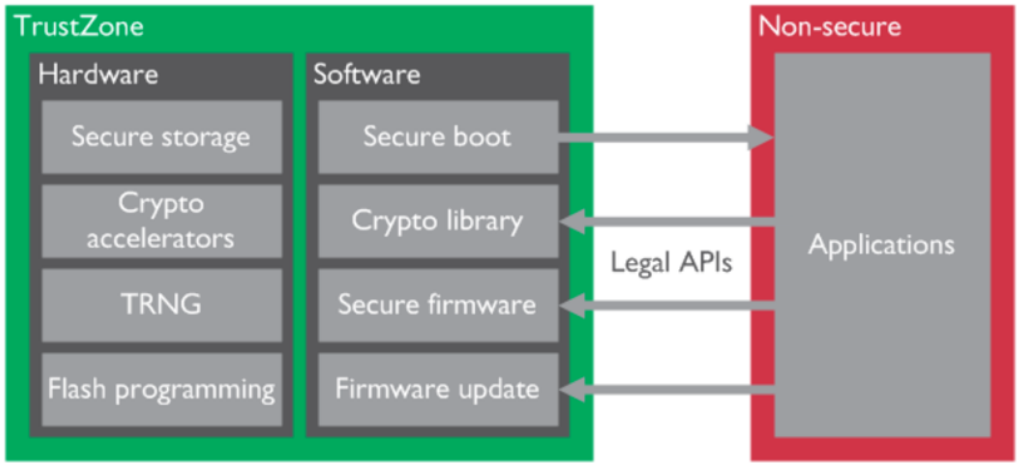
Typically on v7-M your RTOS runs privileged and programs the MPU to provide protection against _accidental_ wrong memory accesses.
PSoC4 tried something in the middle, also using the MPU to restrict firmware securely.

ARMv8-M make this more explicit:
The trusted side is secured by explicit hardware support.



TrustZone is a marketing term. TrustZone-A is not technically similar to TrustZone-M!!!
TrustZone-M is much more direct: Uses a different, hardware-accelerated mechanism for mode switching than on Cortex-A.

Technical term is: Cortex-M Security Extensions: CMSE; -cmse compiler flag
We call the trusted side, “secure side”, untrusted “non-secure”.

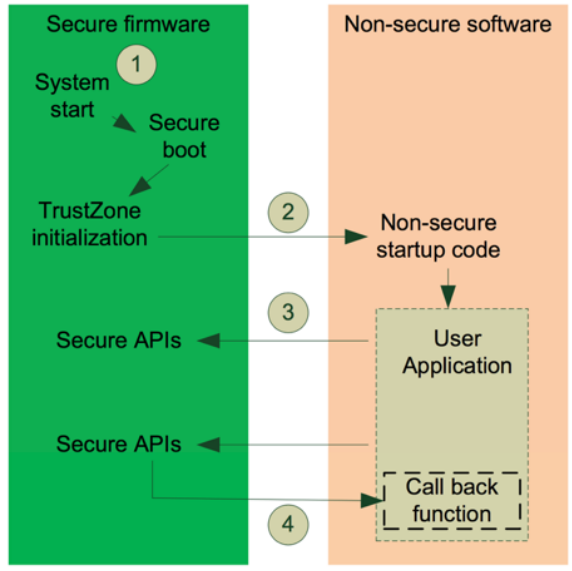


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

The basic idea is to provide secure services on the trusted domain:

- Hey, secure side, please encrypt this memory blob, but don't expose my keys.
- Hey, secure side, please verify this firmware update memory blob and apply it securely.

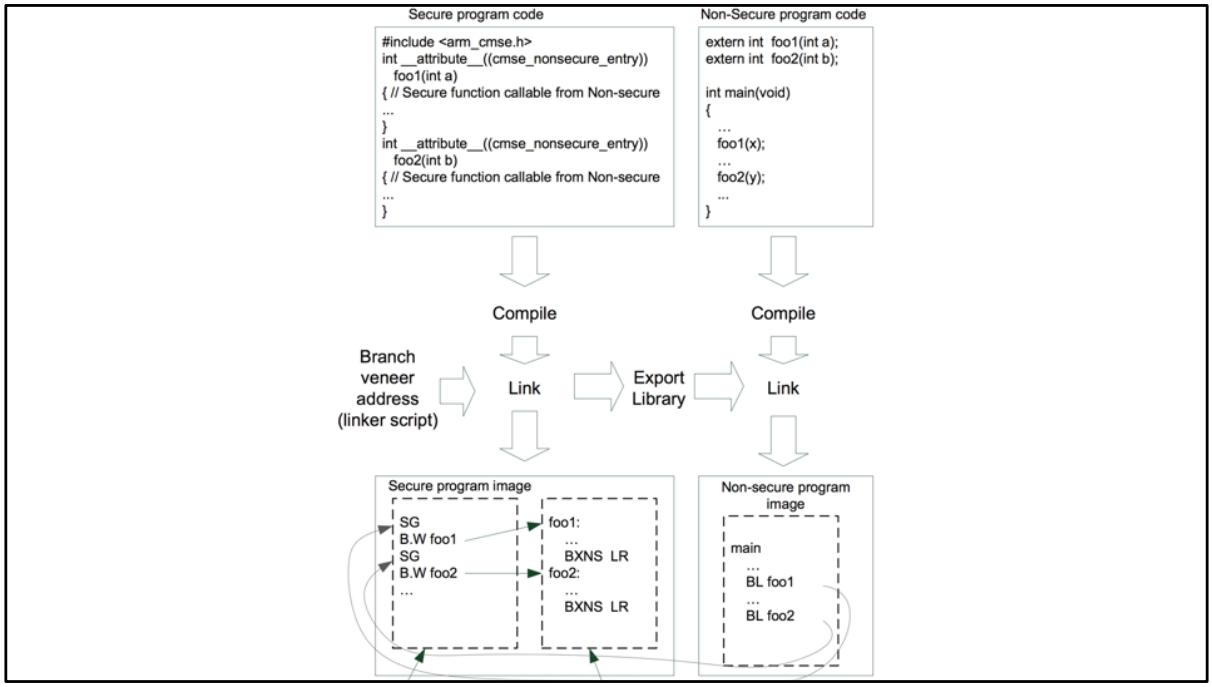
Vendors may ship a device with a TrustZone implementation! Debug access is then limited only to the non-secure side!



"Software Development in ARMv8-M Architecture" by Joseph Yiu

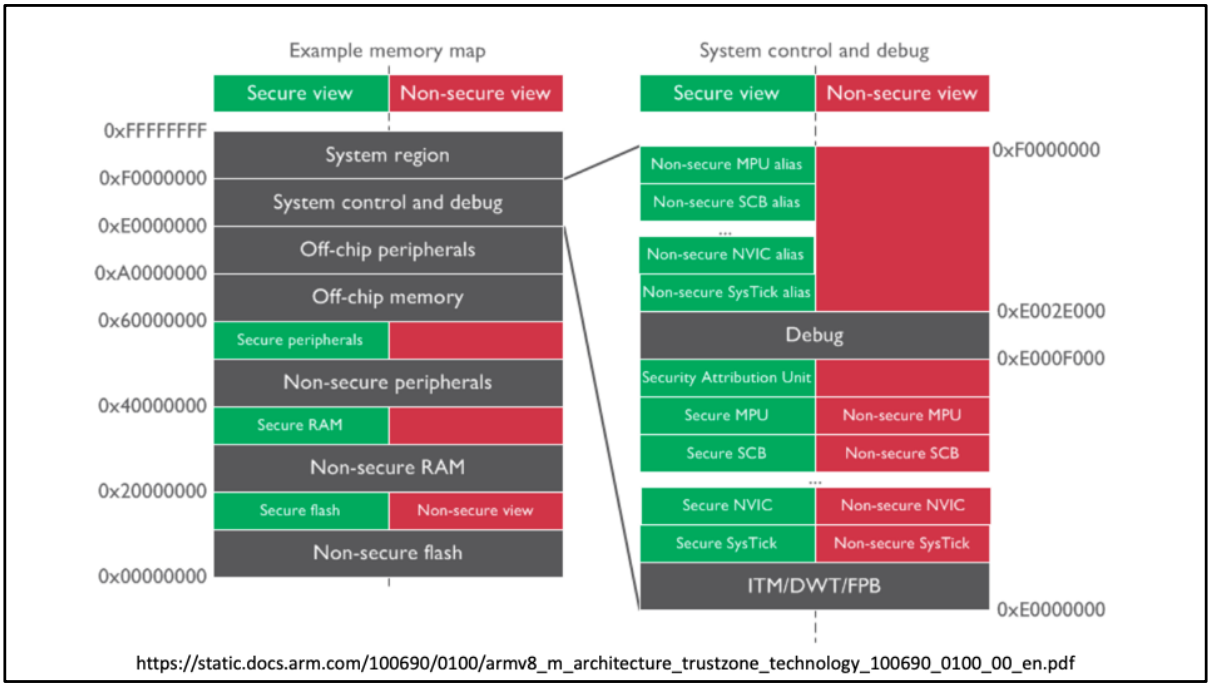
Execution starts on the secure side.

1. Configure the system
2. perhaps perform bootloader tasks,
3. Initialize the TrustZone subsystems
4. Delegate execution to the non-secure side
5. Non-secure side may delegate execution back to secure side,
6. OR secure-side may get secure interrupts
7. Secure side can also call back to non-secure side
8. State is saved on the secure stack.



You are building TWO executables now, with TWO linkerscripts!
Secure APIs are a binary interface, but may wrapped in a static library for easier access.

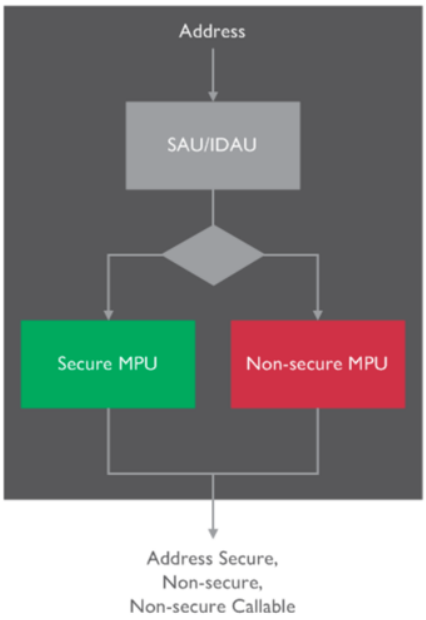
Secure and non-secure have different views on the memory!



Hardware does memory aliasing for you depending on CPU mode.
Non-secure side may not be able to access all flash or ram or peripherals, depending on what the secure side configured.

There are duplicated CPU peripherals:
- 2 Memory Protection Units
- 2 System Control Blocks
- 2 NVICs: Two interrupt vector tables!
- 2 SysTicks: Interrupts are routed to their respective sides.

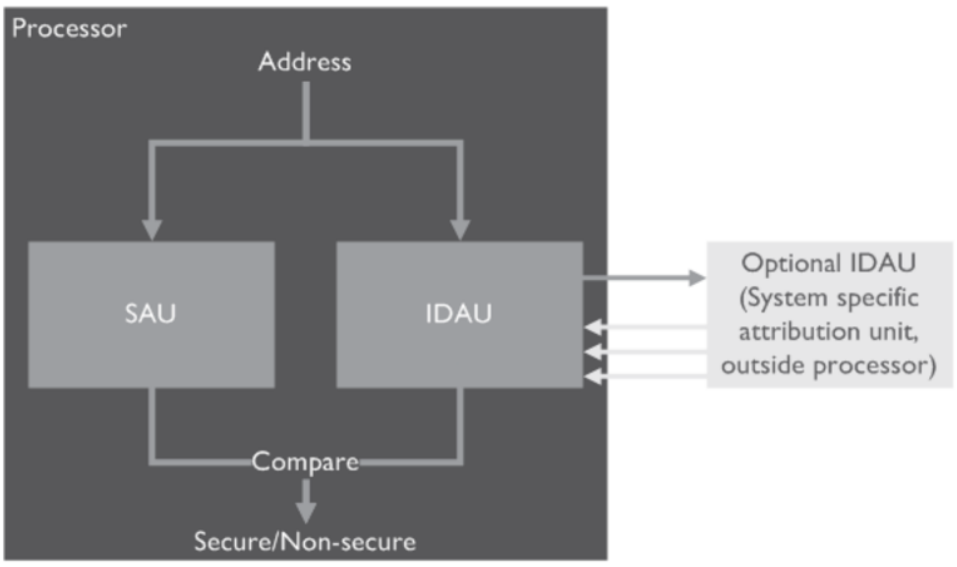
DETAILS ARE VERY IMPLEMENTATION DEFINED!



https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

There are two completely independent MPUs: one for each side.
This is cool, because your RTOS can now run completely independently in the non-secure side.
It has it's own MPU (for safety), it's own SysTick, some fault handler interrupts.

BUT: We need a new peripheral to attribute which memory is secure/non-secure:
The SAU: Security Attribution Unit is queried first, and memory access compared with CPU mode.
(Sau means pig in German *giggle*)

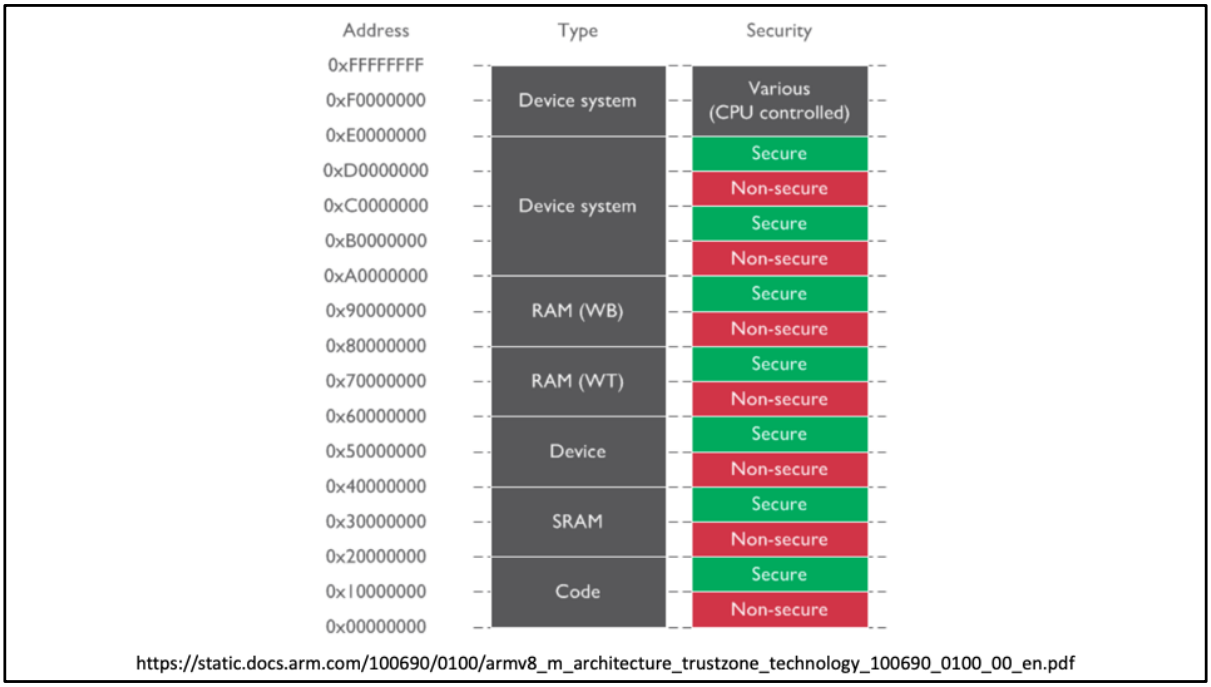


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

SAU is configurable at runtime, virtually identical use as the MPU.
Use this to split up your memories, assign peripherals to sides, etc.

IDAU: (Implementation-Defined Attribution Unit) is provided non-mutable background attribution map implemented by the vendors in hardware for their specific chip implementation.
It predefines secure and non-secure aliases, to save you from programming that into the SAU yourself.

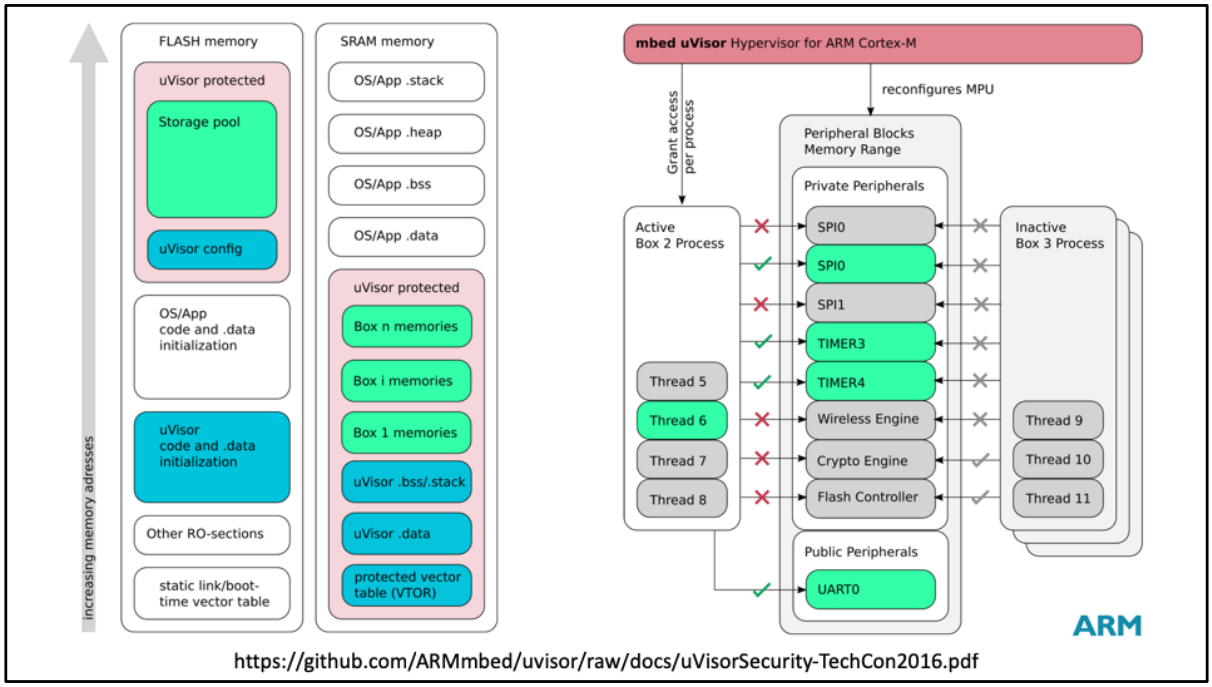
If SAU regions overlap or the IDAU comparison fails => Always attributes memory as secure.



https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

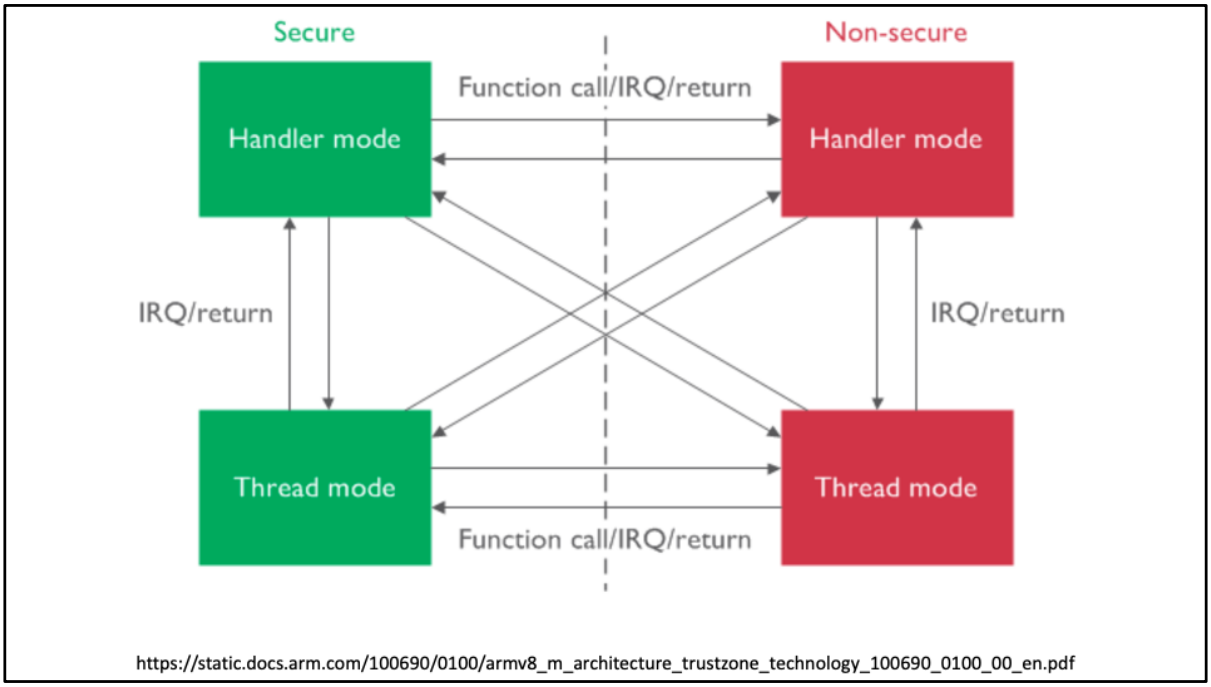
The secure side can only access memory attributed as secure.
The non-secure side can only access memory attributed as non-secure.

So the aliasing helps with the attribution, by splitting the memory map in half. Bit 28 is used by ARM as an example, so 256MB chunks are aliased. IMPLEMENTATION DEFINED.



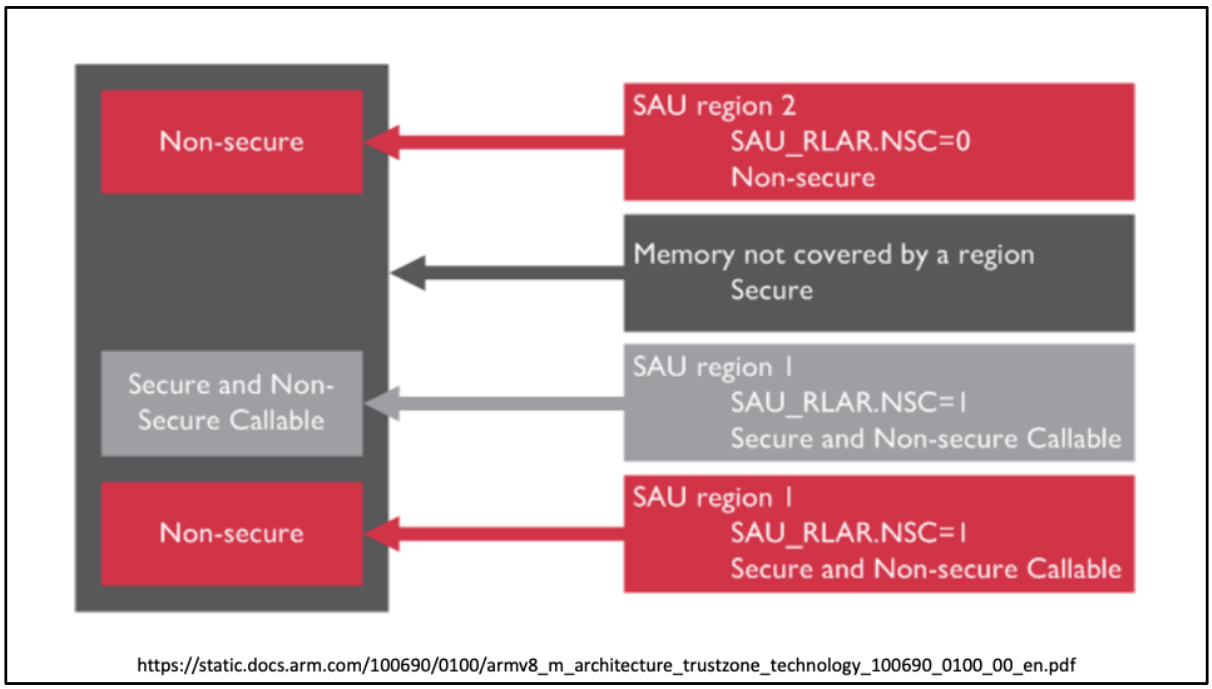
https://github.com/ARMmbed/uvisor/raw/docs/uVisorSecurity-TechCon2016.pdf

Example configuration in uVisor: We implemented mutually distrustful domains, so ”multiple non-secure environments”.
We reprogram the SAU/MPU on environment switch (also runs on ARMv7-M).



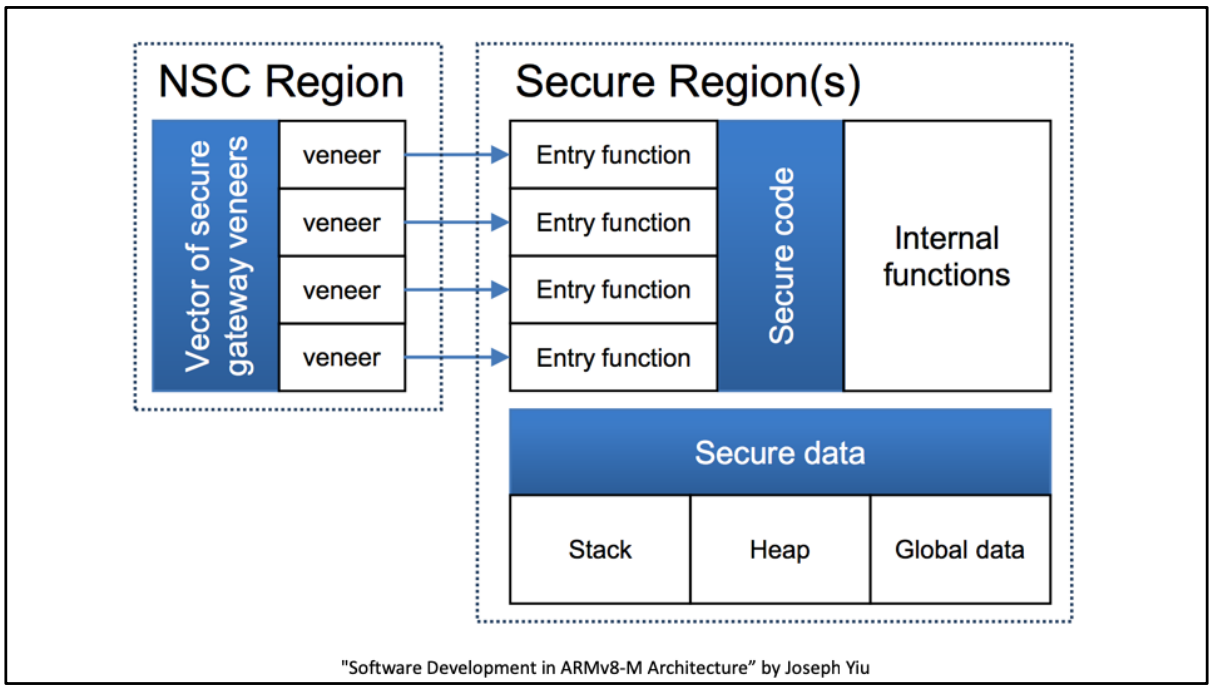
So how do we actually call a secure API?
IT'S SIMPLE REALLY, JUST LEARN ABOUT THESE 12 TRANSITION TYPES. LOL.

But how do you call into the Secure side, if you cannot access this memory at all?

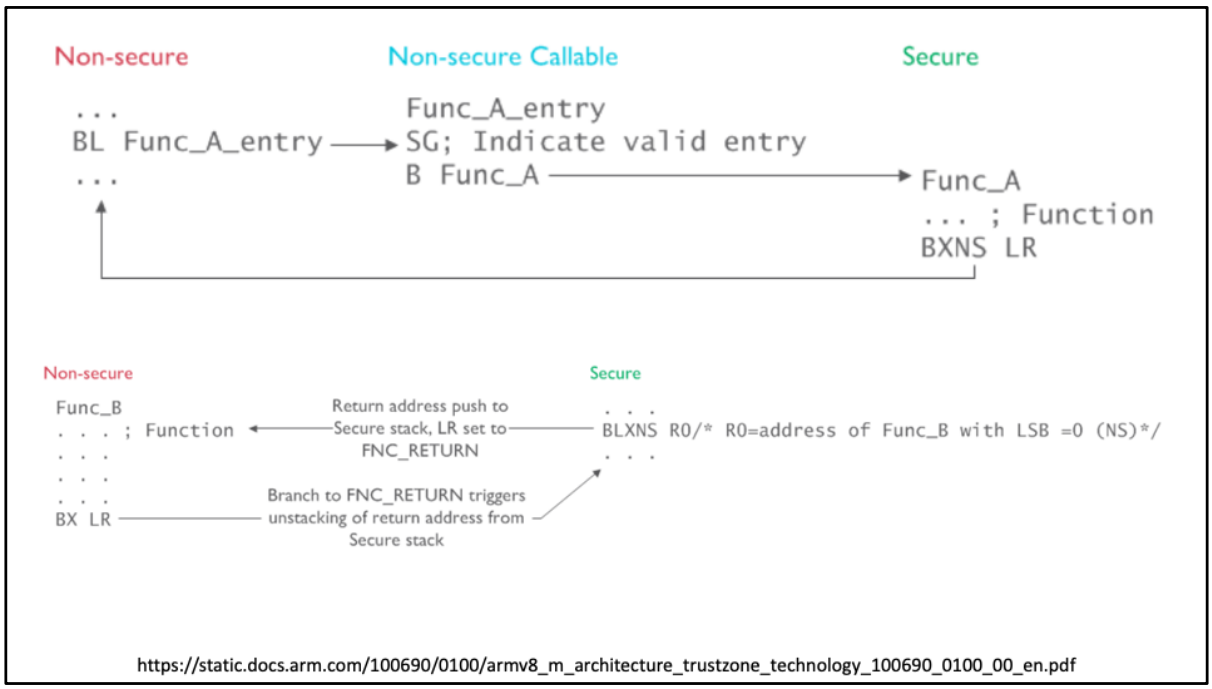


You introduce a THIRD memory attribute: Secure and Non-Secure Callable.

This memory is secure, so you cannot read it (aka. data fetch), but the non-secure side can call into it, (aka. instruction fetch).



You need to be able to control the entry point into the secure side, otherwise you can reconstruct some secure side instructions by observing CPU side effects of its execution.
So you actually call a veneer in the NSC region which simply forwards to the secure side

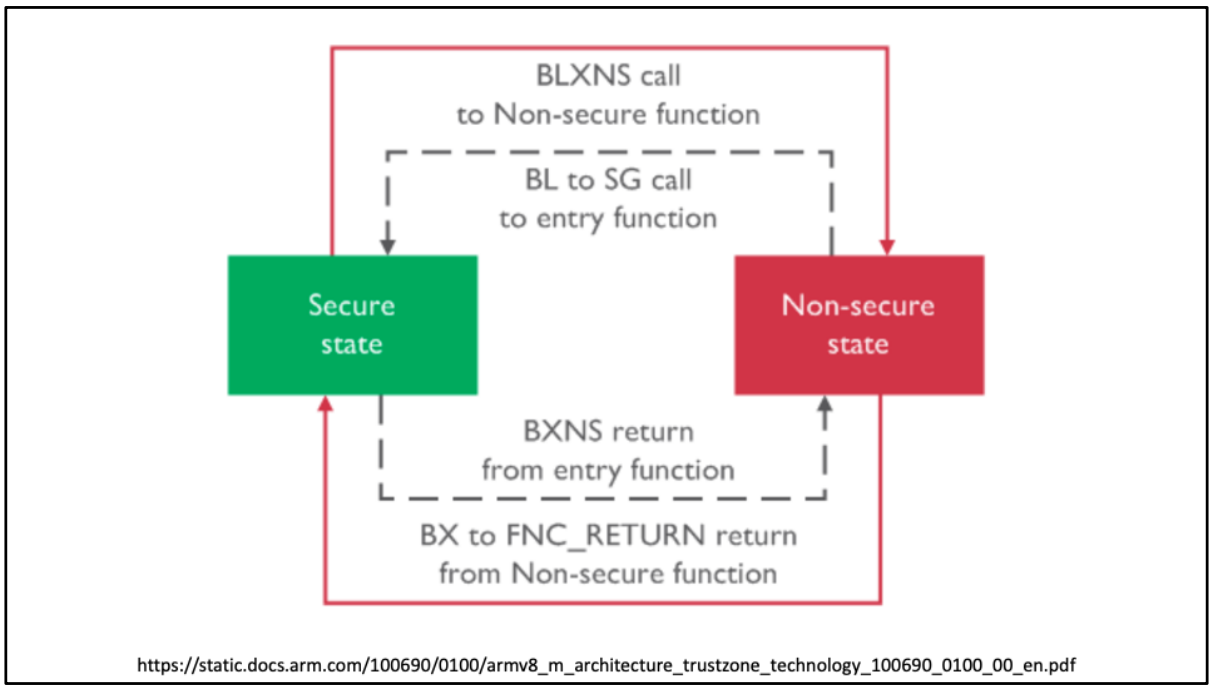


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

On call to a NSC region from NS side, the CPU will fetch ONE 16-bit instruction from the NSC side and if it's not a SecureGateway instruction, it will SecureFault. (Function Call is not in sudoers file, THIS INCIDENT WILL BE REPORTED).
The CPU mode transition happens on the SG instruction and then you can just call into the secure code.
To go back, branch to NS with link register.

(SG instructions are NOP when executed by the secure side.)
SG instruction encoding may also occur accidentally in normal code or data, The NSC region may ONLY contain veneer code!

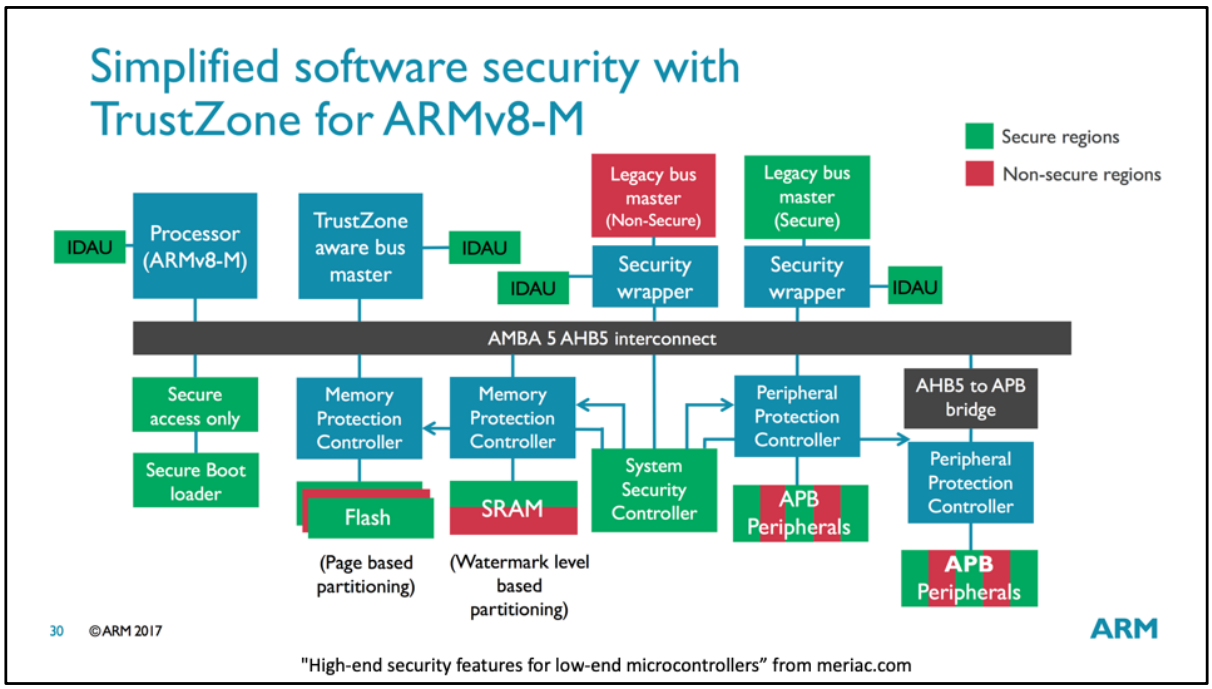
When the secure side calls the NS side, the link register contains FNC_RETURN which obfuscuates the caller address to the NS side!



https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

These are the “lightweight” transitions, they are similar in cost to a function call rather than a synchronous interrupt (aka re-privileging via SVC).

I’m not going into the details of cross-side interrupt handling, but there is extended support for pushing secure-side registers for mode changes in tail-chaining, securing FPUs registers.
Worst case stack frame can get up to 212B large!



"High-end security features for low-end microcontrollers" from meriac.com

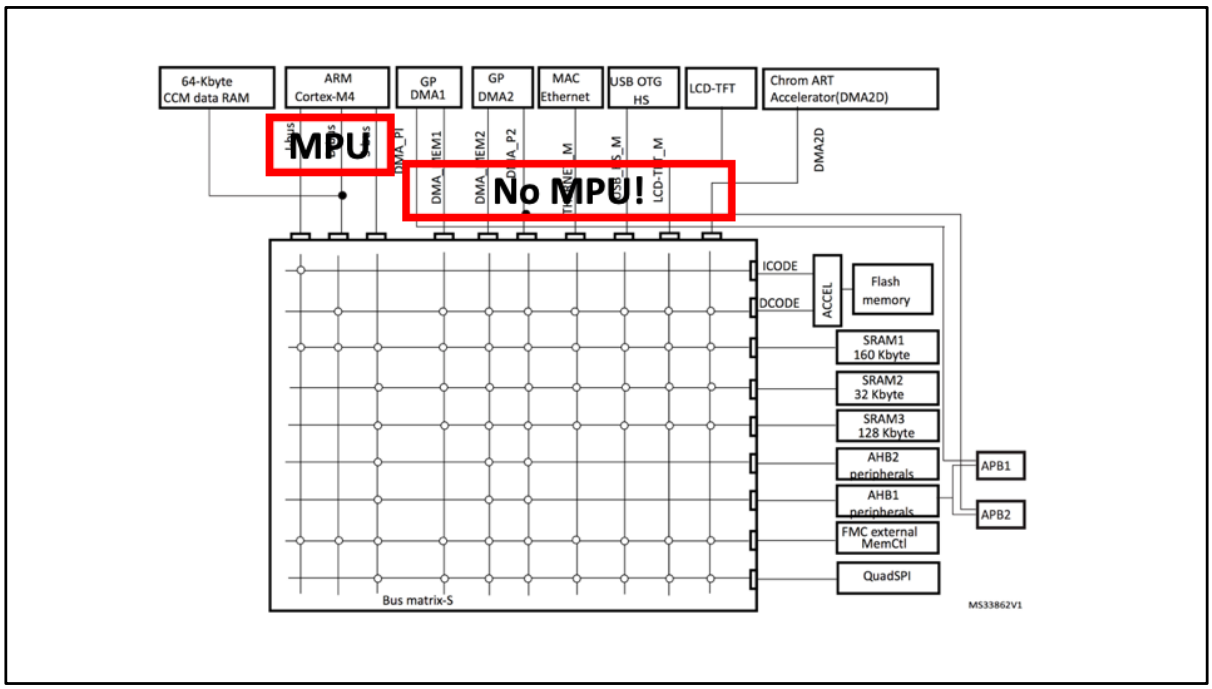
ARMv8-M also extends security attributes into the Busses with the new AHB5 specification.
This means DMA transfers is also bound to security attribution.

Thank you for listening!

Niklas Hauser: @salkinium (salkinium.com)
Milosch Meriac: @FoolsDelight (meriac.com)

uVisor: github.com/ARMmbed/uVisor
ARM PSA: Platform Security Architecture

uVisor is part of the PSA, which is the official architecture from ARM for consolidating security for TrustZone on Cortex-A and Cortex-M.



Why is MPU not enough to protect on ARMv7-M? Because MPU only protects accesses of the Cortex-M CPU!!!!
ANY other bus master (like all DMAs) have complete and uncontrolled access to the whole memory map.