Introduction to ARMv8-M and TrustZone-M

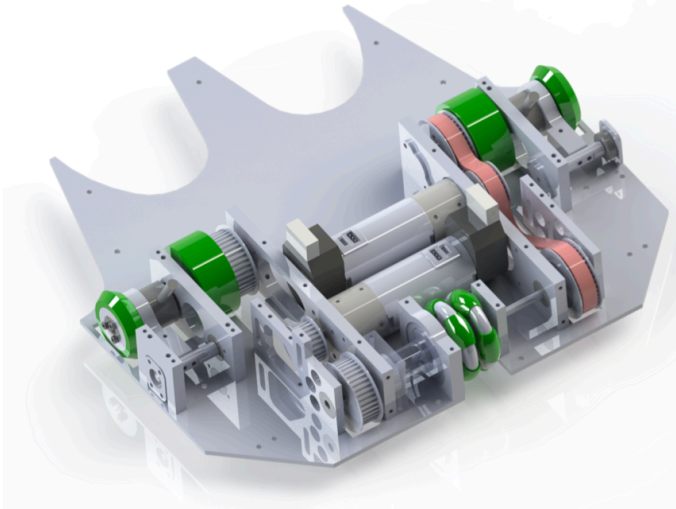
Niklas Hauser

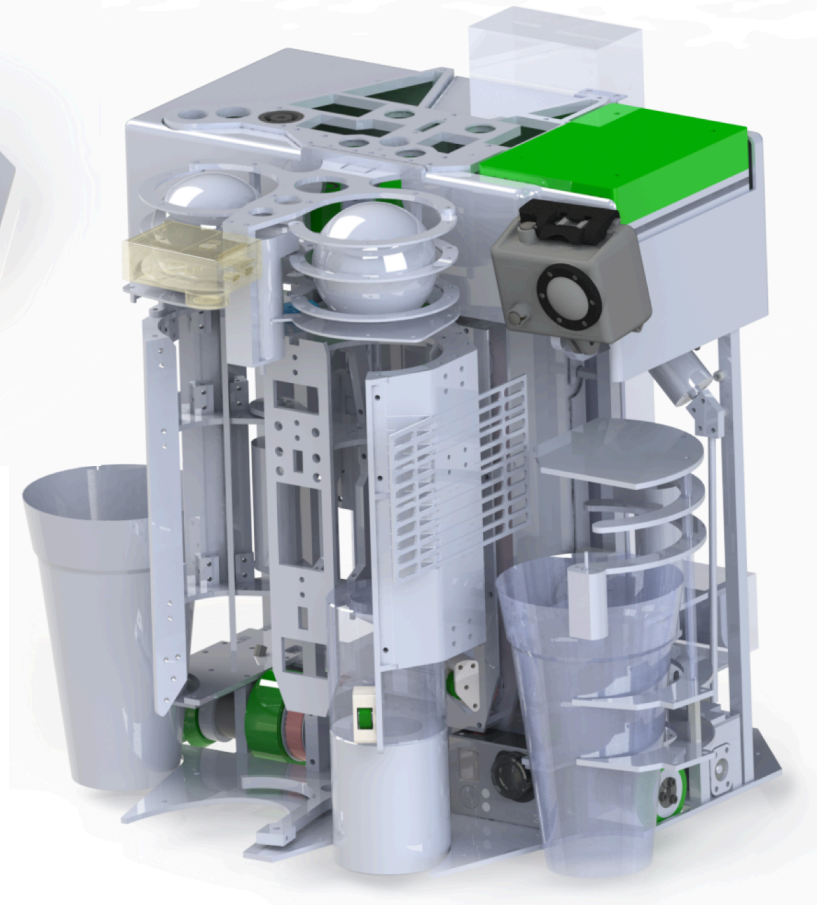
emBO++ 2018


RWTH AACHEN
UNIVERSITY

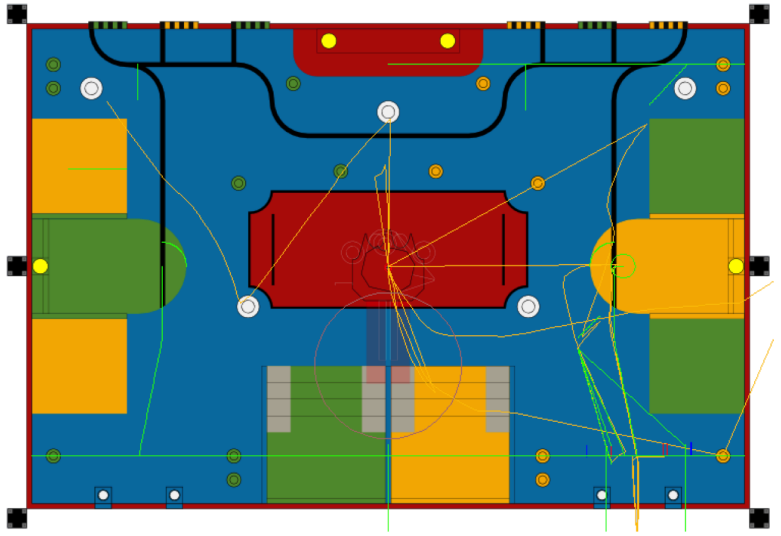
RCA

Roboterclub Aachen e.V.

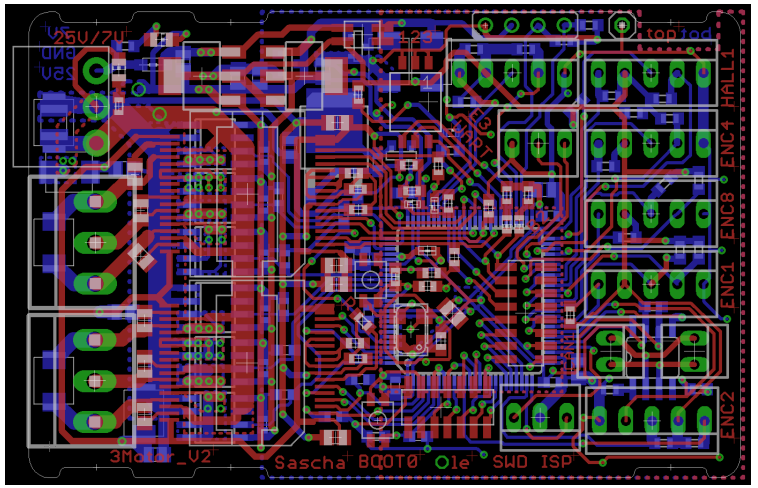




Mechanical Engineering

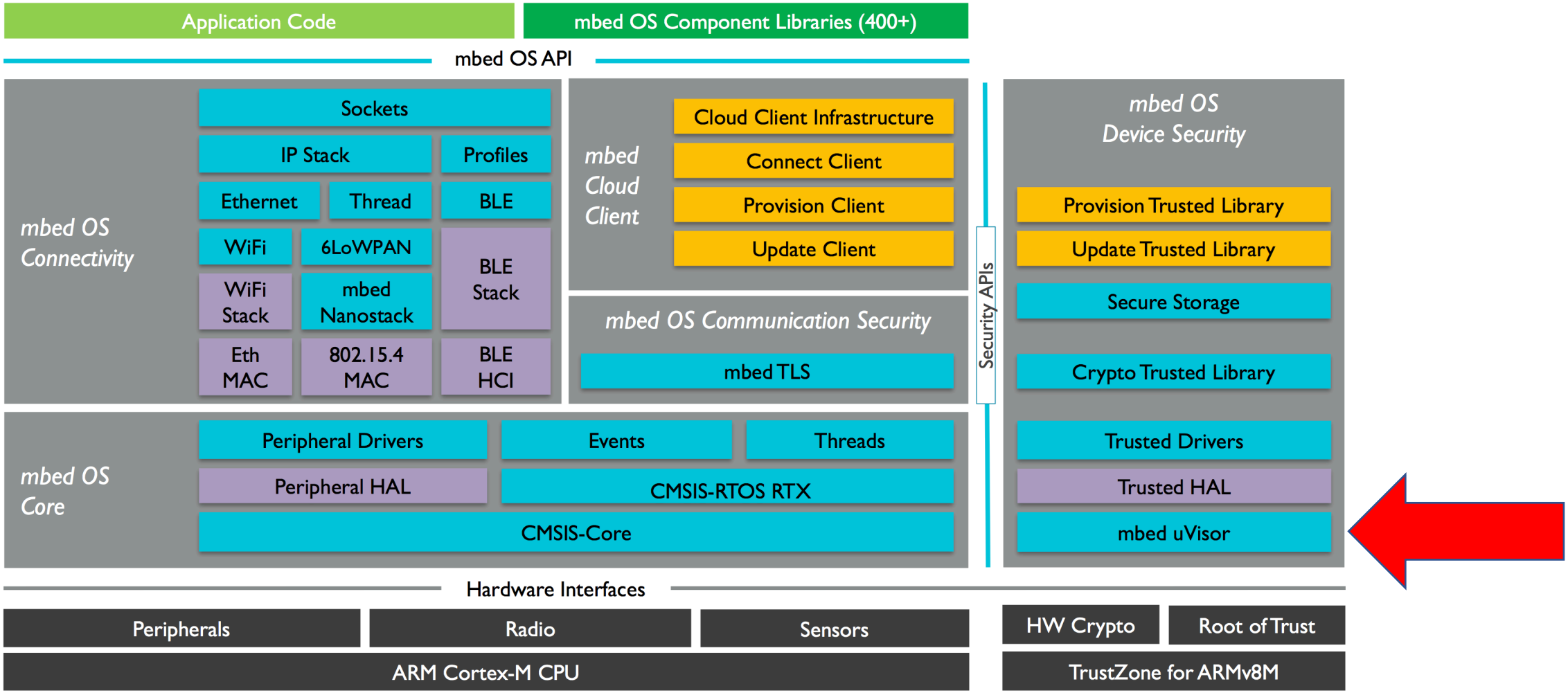


Software Engineering



Electronic Engineering

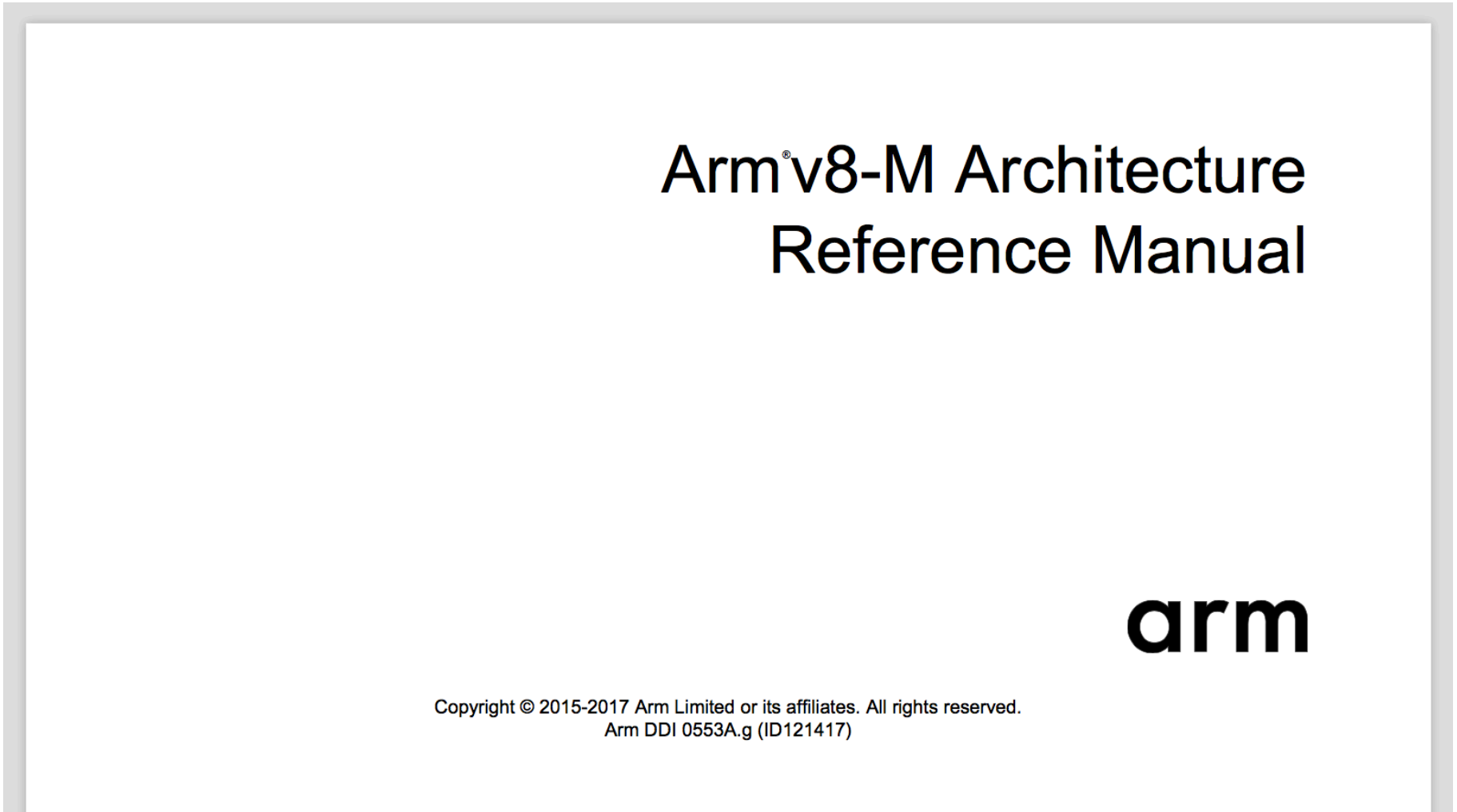
ARM mbed OS: architected platform security



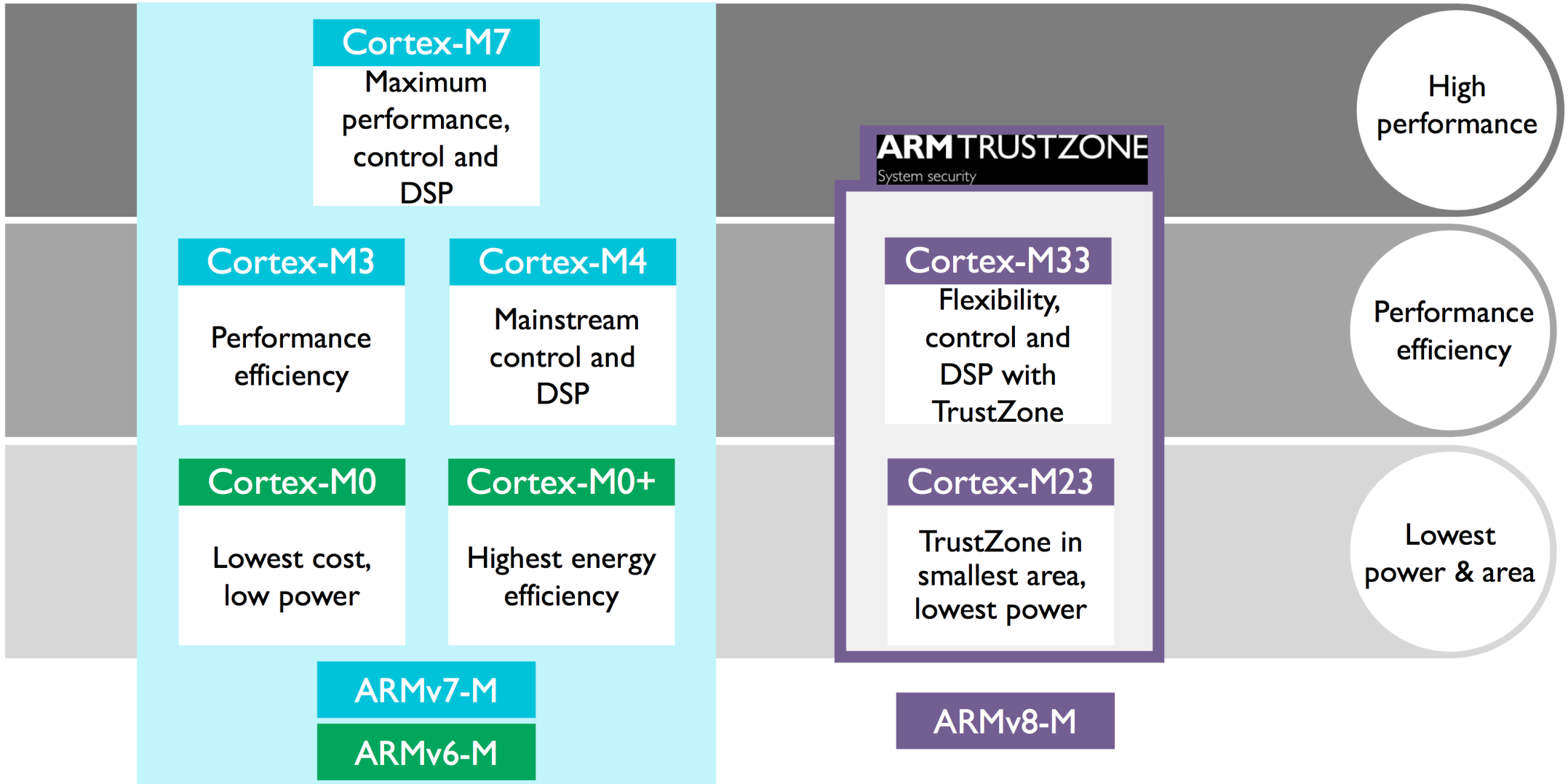
ARM

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

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

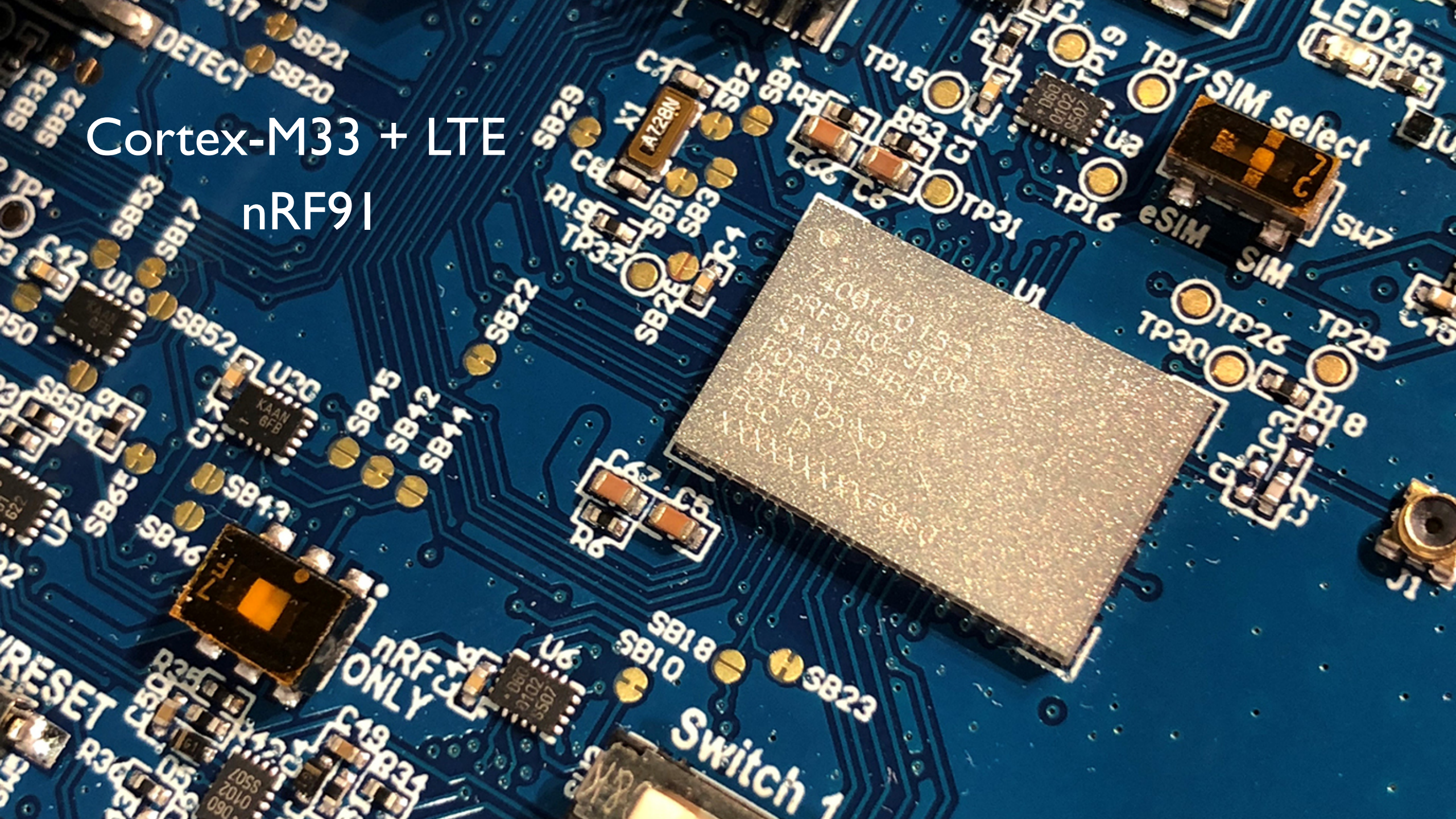


Bringing TrustZone to the Cortex-M family



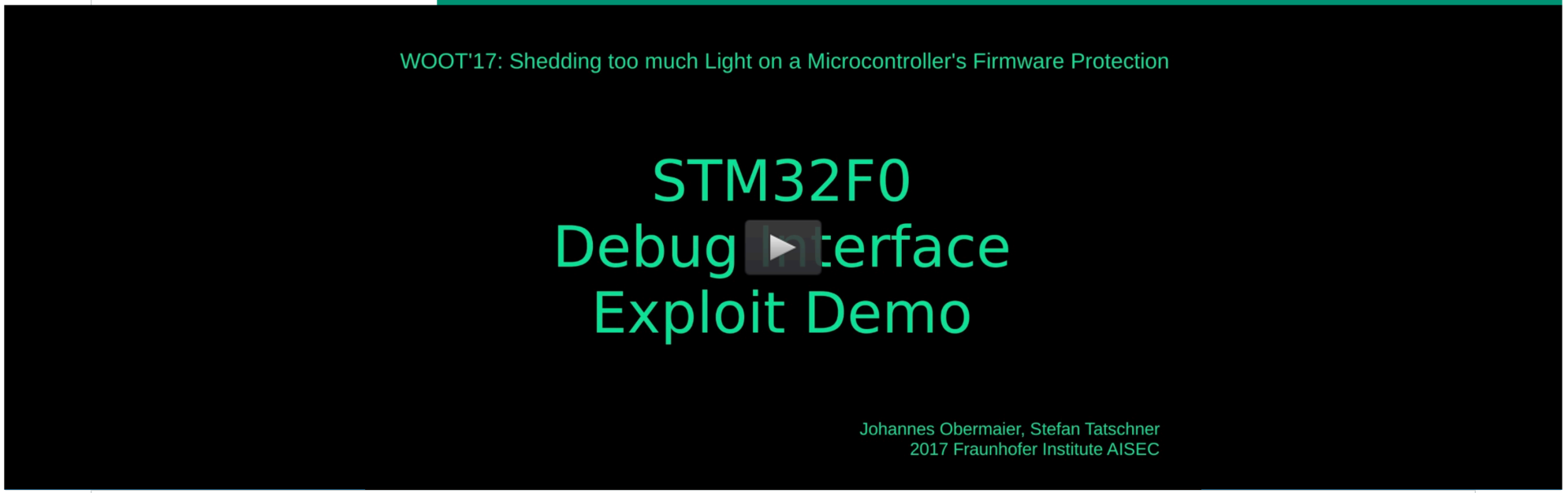
ARM

https://www2.keil.com/docs/default-source/default-document-library/using_trustzone_on_arm_cortex-m23_and_cortex-m33.pdf


Cortex-M33 + LTE
nRF91
SIM select
eSIM
SIM
Switch 1
nRF ONLY
RESET
DETECT

Fraunhofer AISEC

Fraunhofer Institute for Applied and Integrated Security



Shedding too much Light on a Microcontroller's Firmware Protection

Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT '17) - Vancouver, BC, Canada

Almost every microcontroller with integrated flash features firmware readout protection. This is a form of content protection which aims at securing intellectual property (IP) as well as cryptographic keys and algorithms from an adversary. One series of microcontrollers are the STM32 which have recently gained popularity and thus are increasingly under attack. However, no practical experience and information on the resilience of STM32

Contact

Johannes Obermaier

Archives

Thursday, November 5, 2015

Firmware dumping technique for an ARM Cortex-M0 SoC

by Kris Brosch

One of the first major goals when reversing a new piece of hardware is getting a copy of the firmware. Once you have access to the firmware, you can reverse engineer it by disassembling the machine code.

Sometimes you can get access to the firmware without touching the hardware, by downloading a firmware update file for example. More often, you need to interact with the chip where the firmware is stored. If the chip has a debug port that is accessible, it may allow you to read the firmware through that interface. However, most modern chips have security features that when enabled, prevent firmware from being read through the debugging interface. In these situations, you may have to resort to decapping the chip, or introducing glitches into the hardware logic by manipulating inputs such as power or clock sources and leveraging the resulting behavior to successfully bypass these security implementations.

This blog post is a discussion of a new technique that we've created to dump the firmware stored on a particular Bluetooth system-on-chip (SoC), and how we bypassed that chip's security features to do so by only using the debugging interface of the chip. We believe this technique is a vulnerability in the

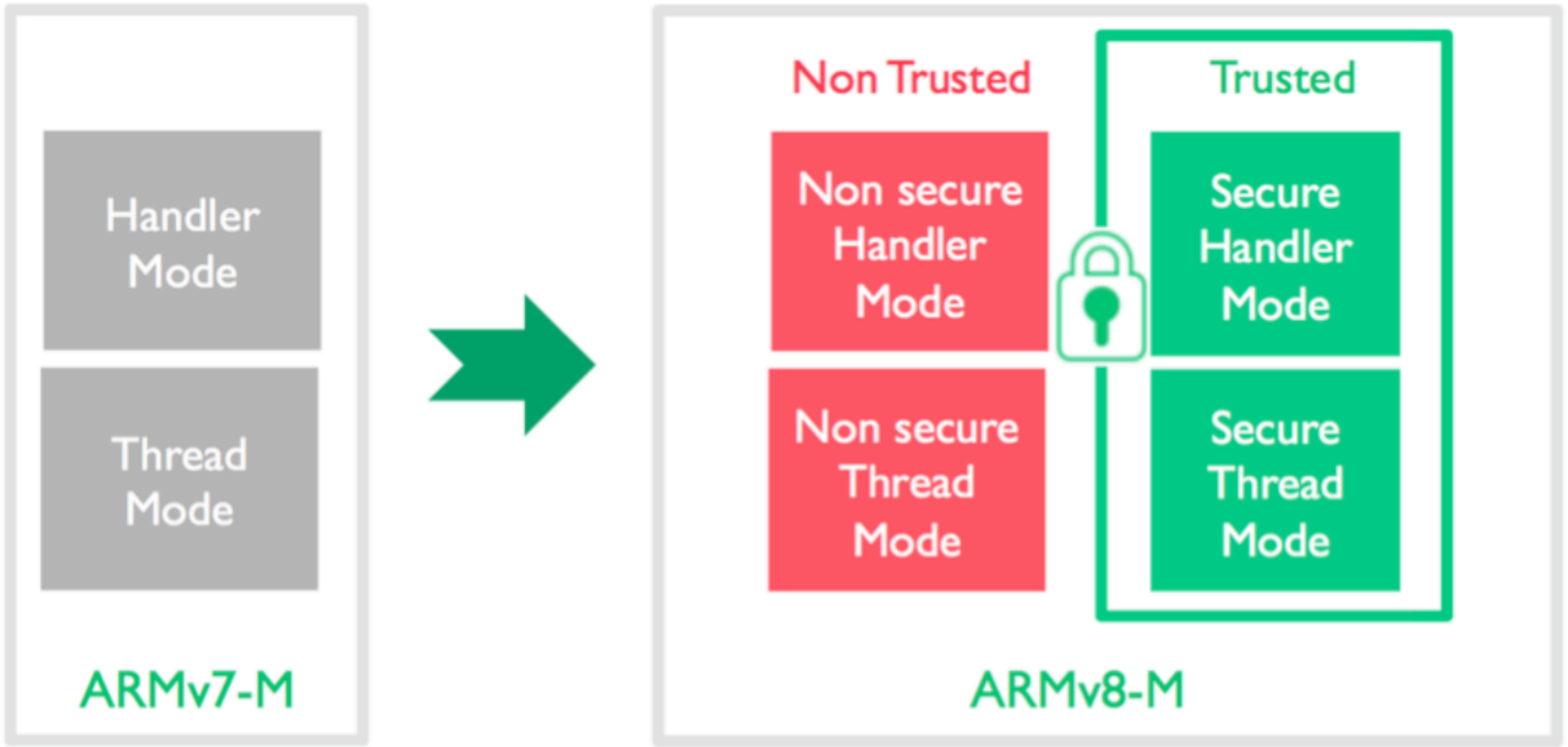
Part 0 - TLDR

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.

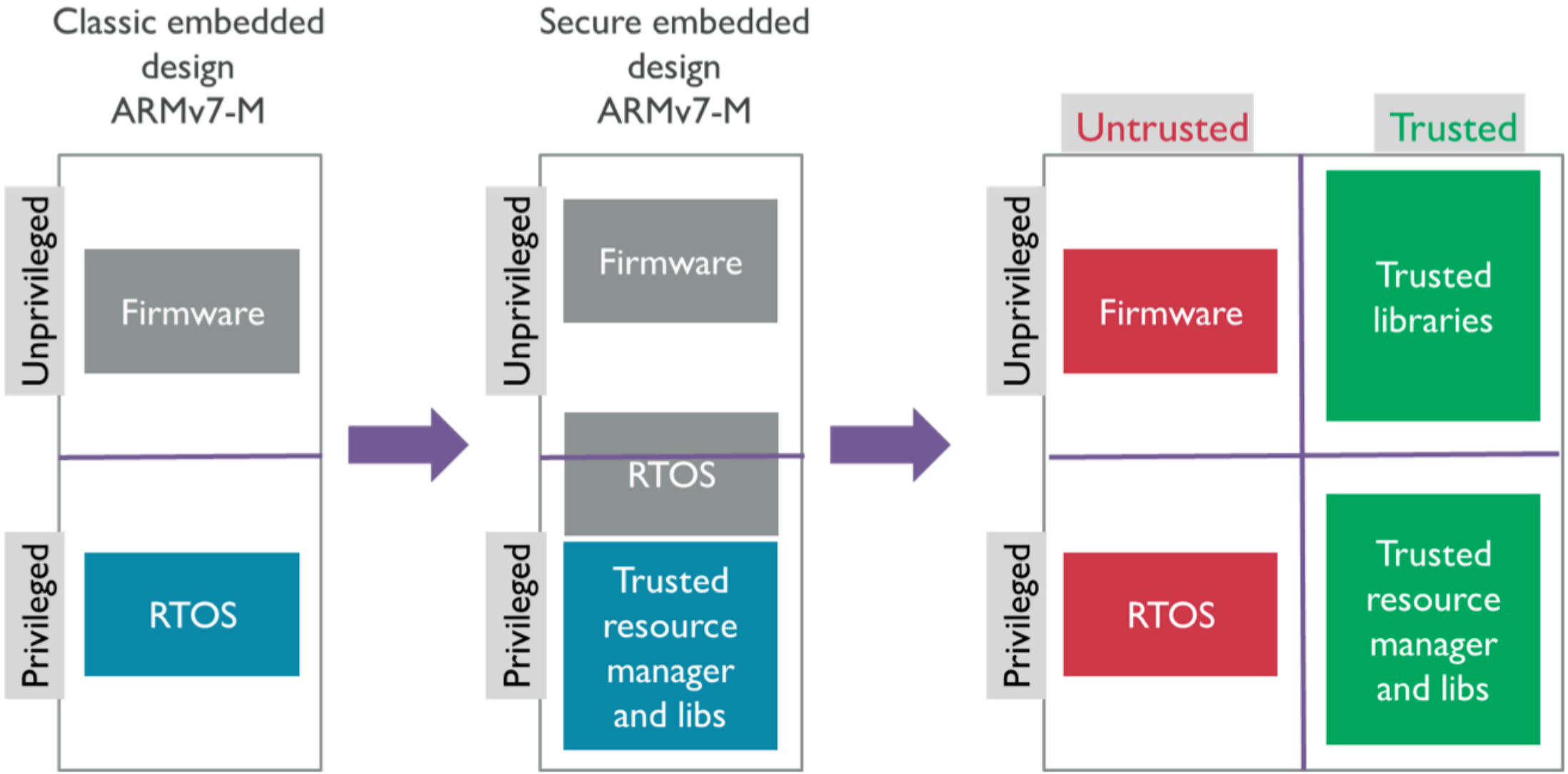
Part 1, where we meet our opponent and look deeply into its eyes...

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

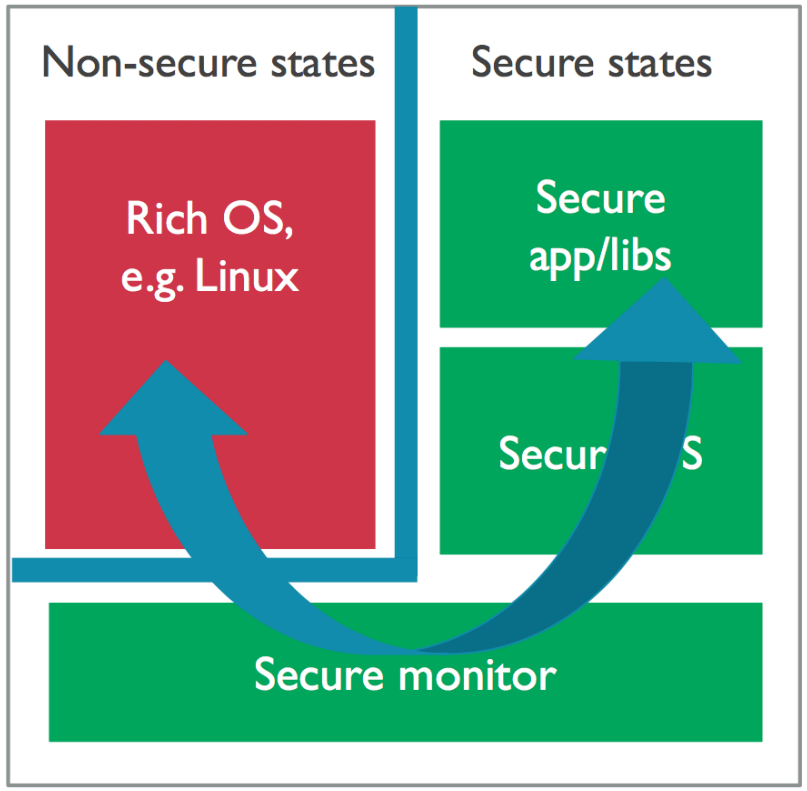


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

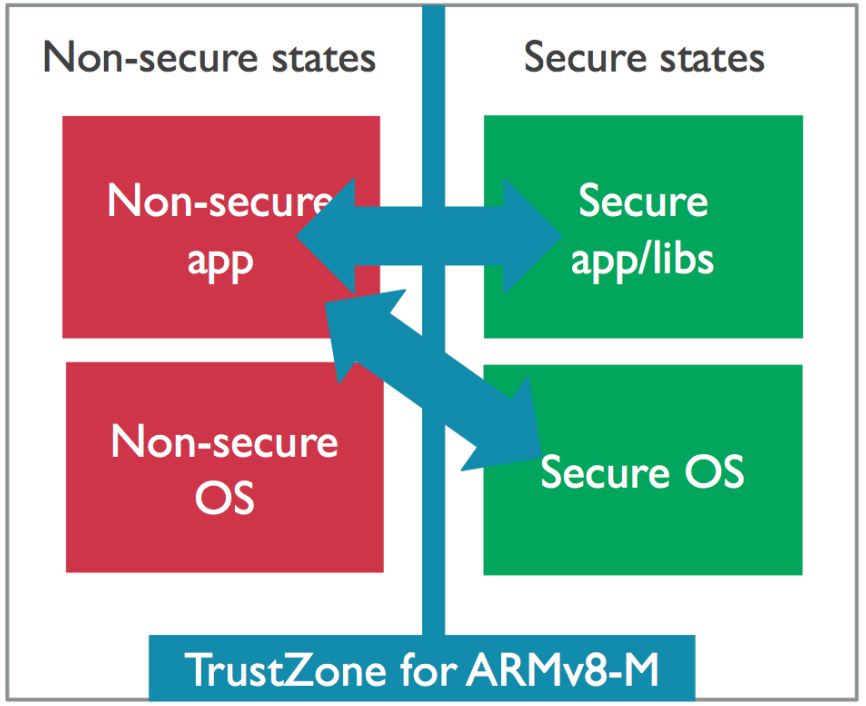


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

TrustZone for ARMv8-A



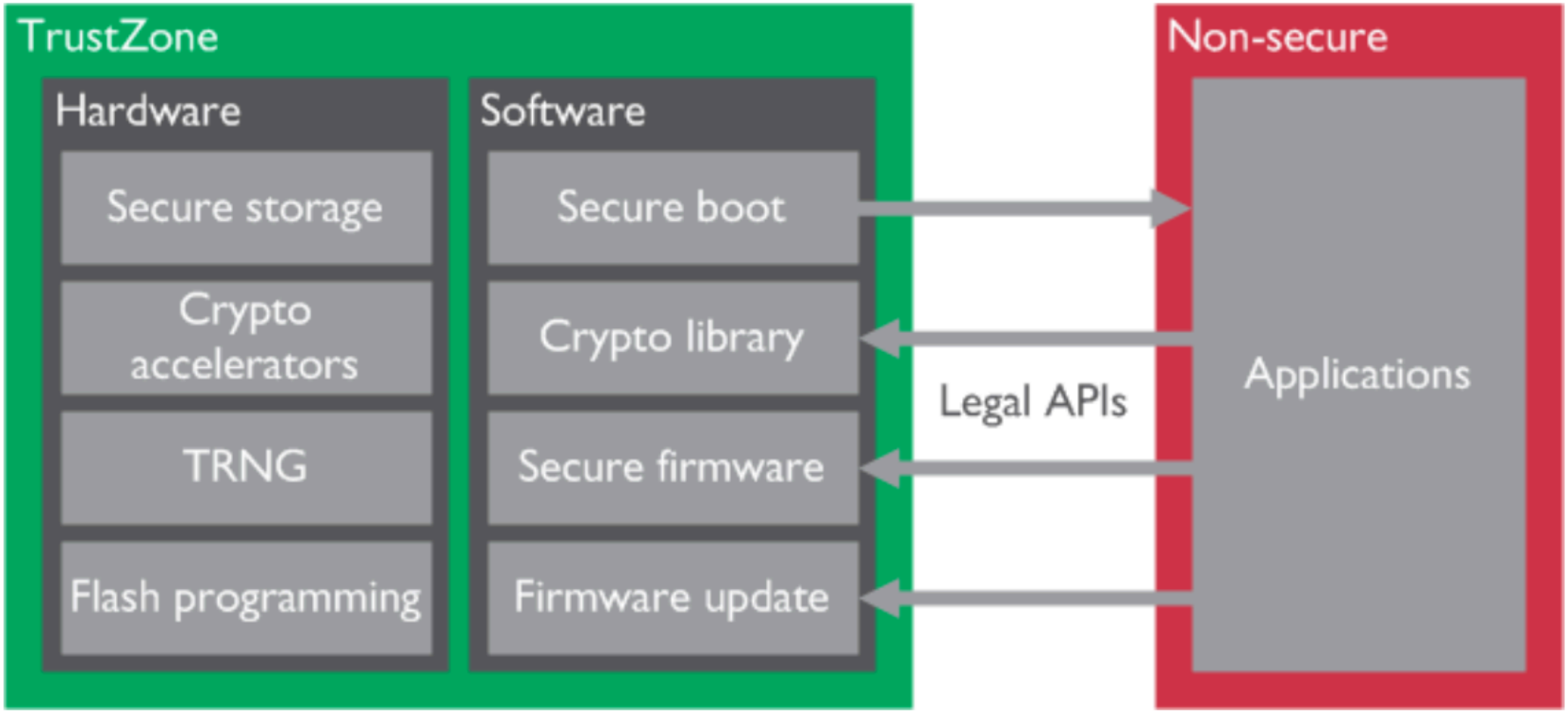
TrustZone for ARMv8-M



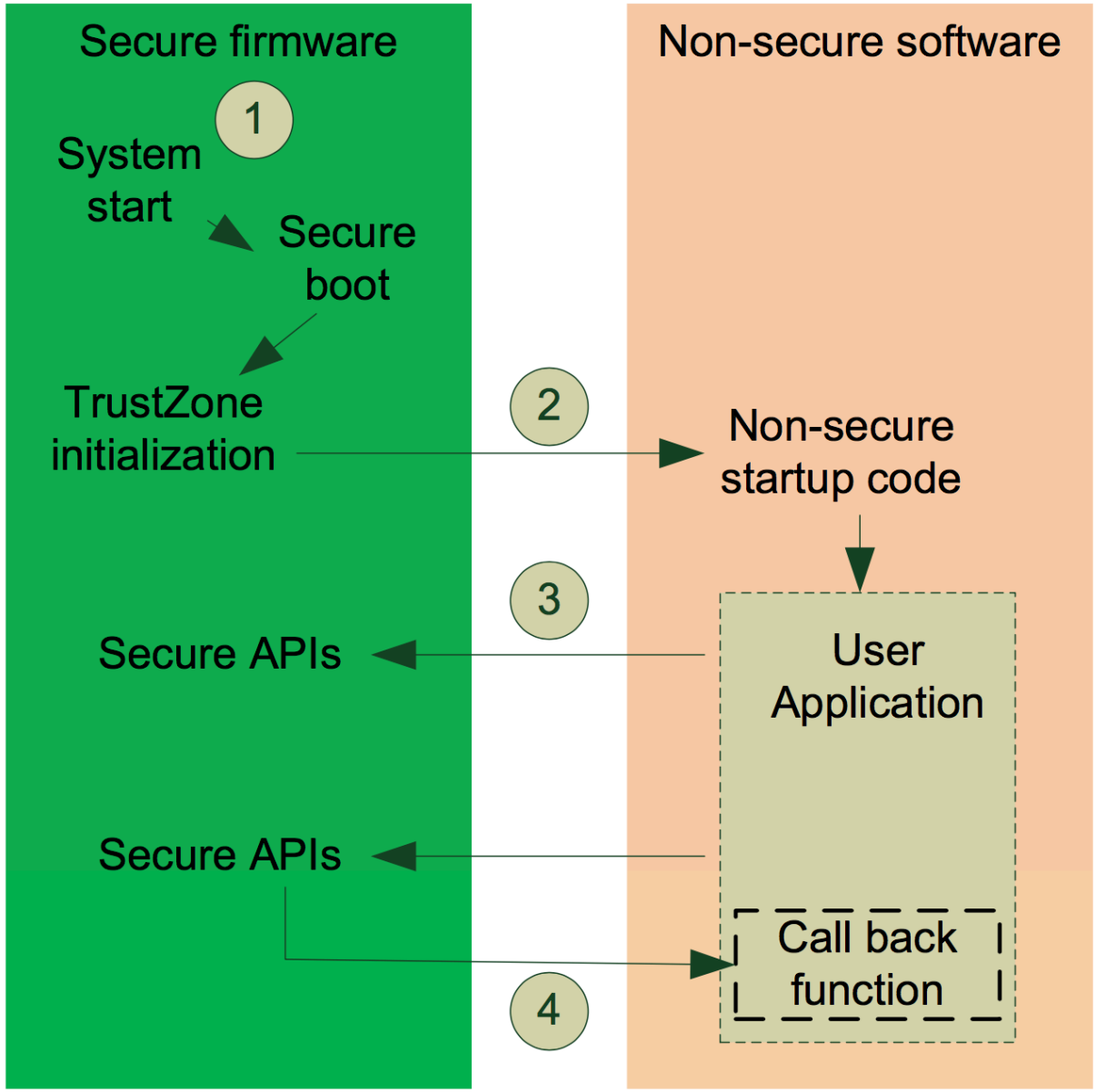
Secure transitions handled by the processor
to maintain embedded class latency

ARM

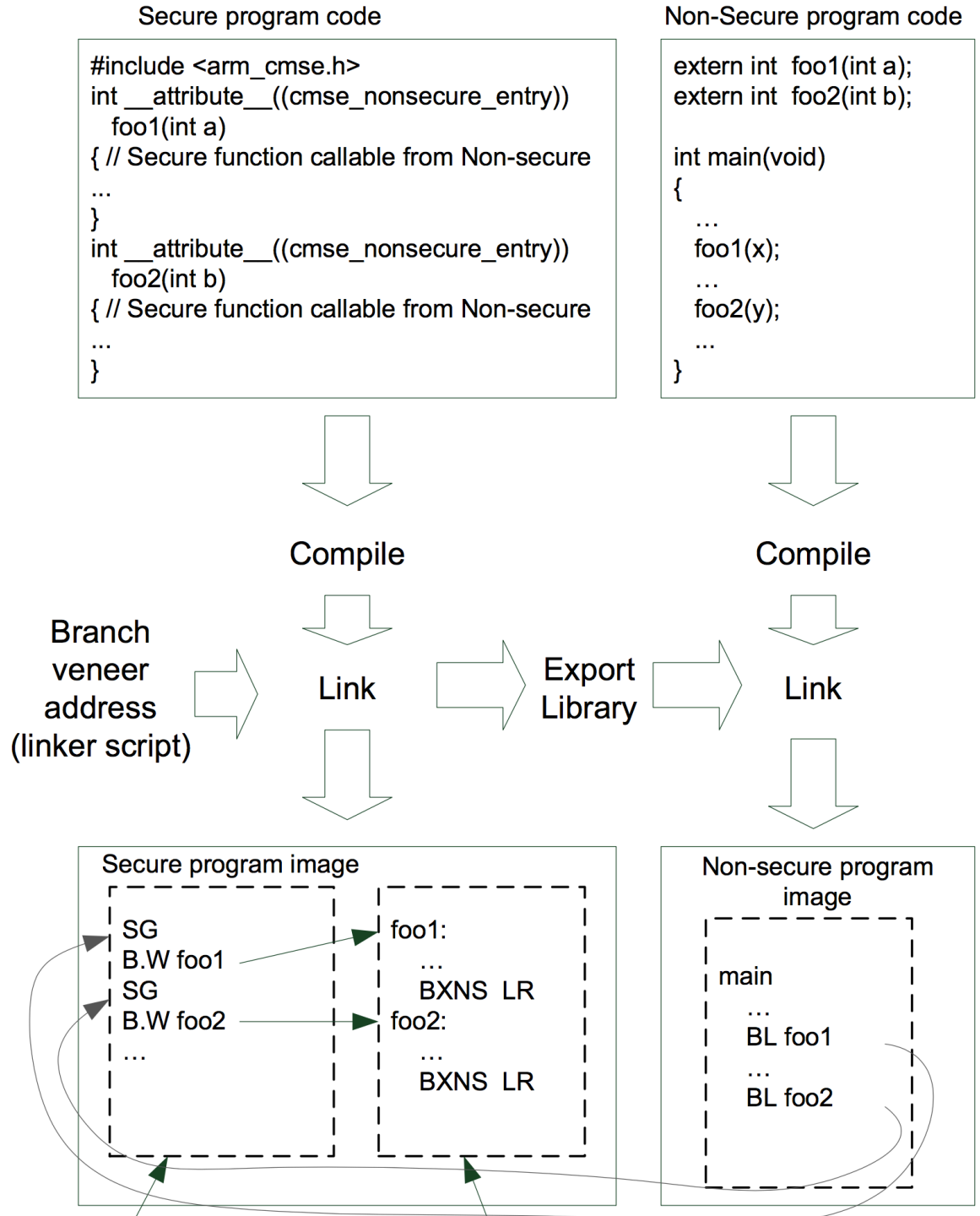
https://www2.keil.com/docs/default-source/default-document-library/using_trustzone_on_arm_cortex-m23_and_cortex-m33.pdf

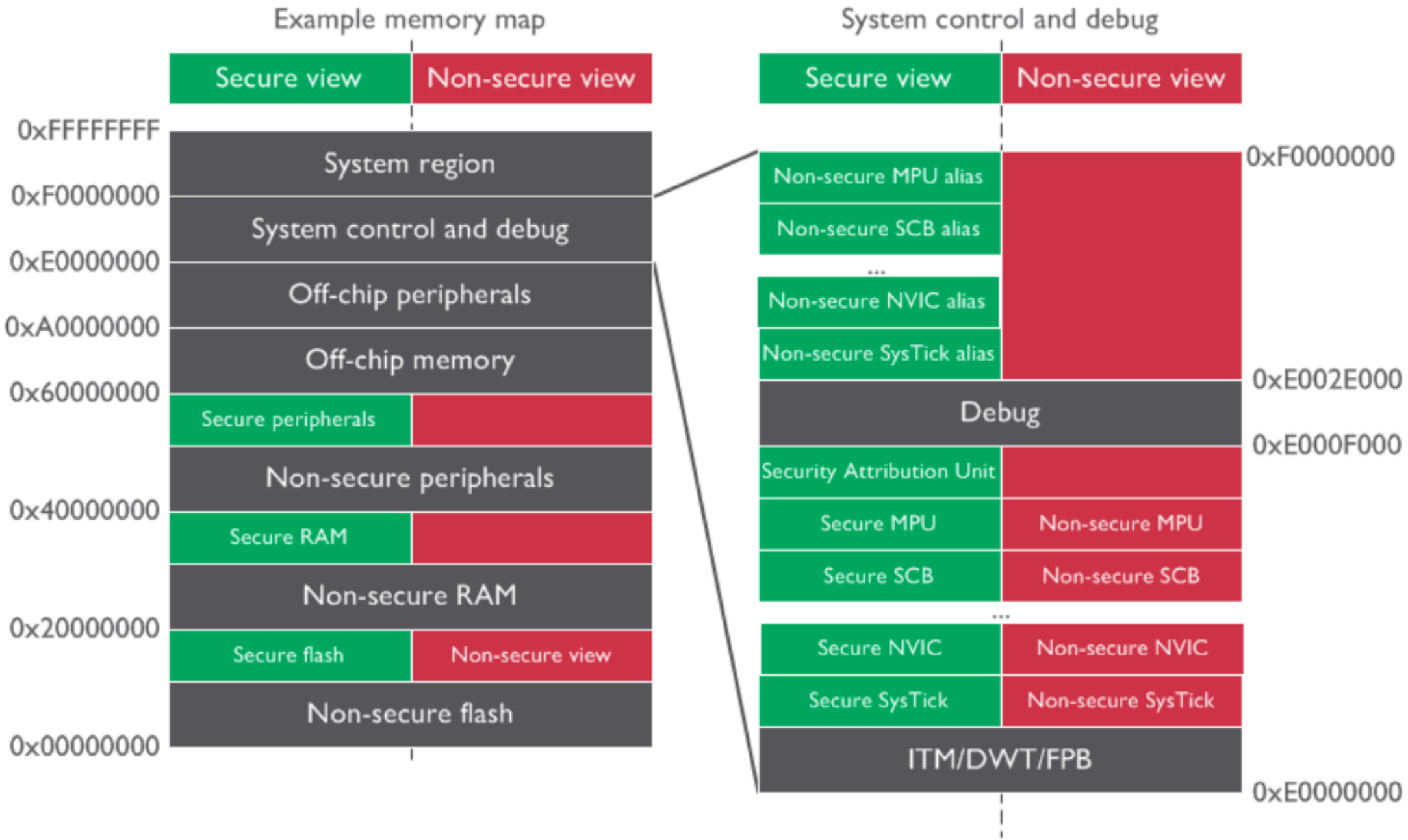


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

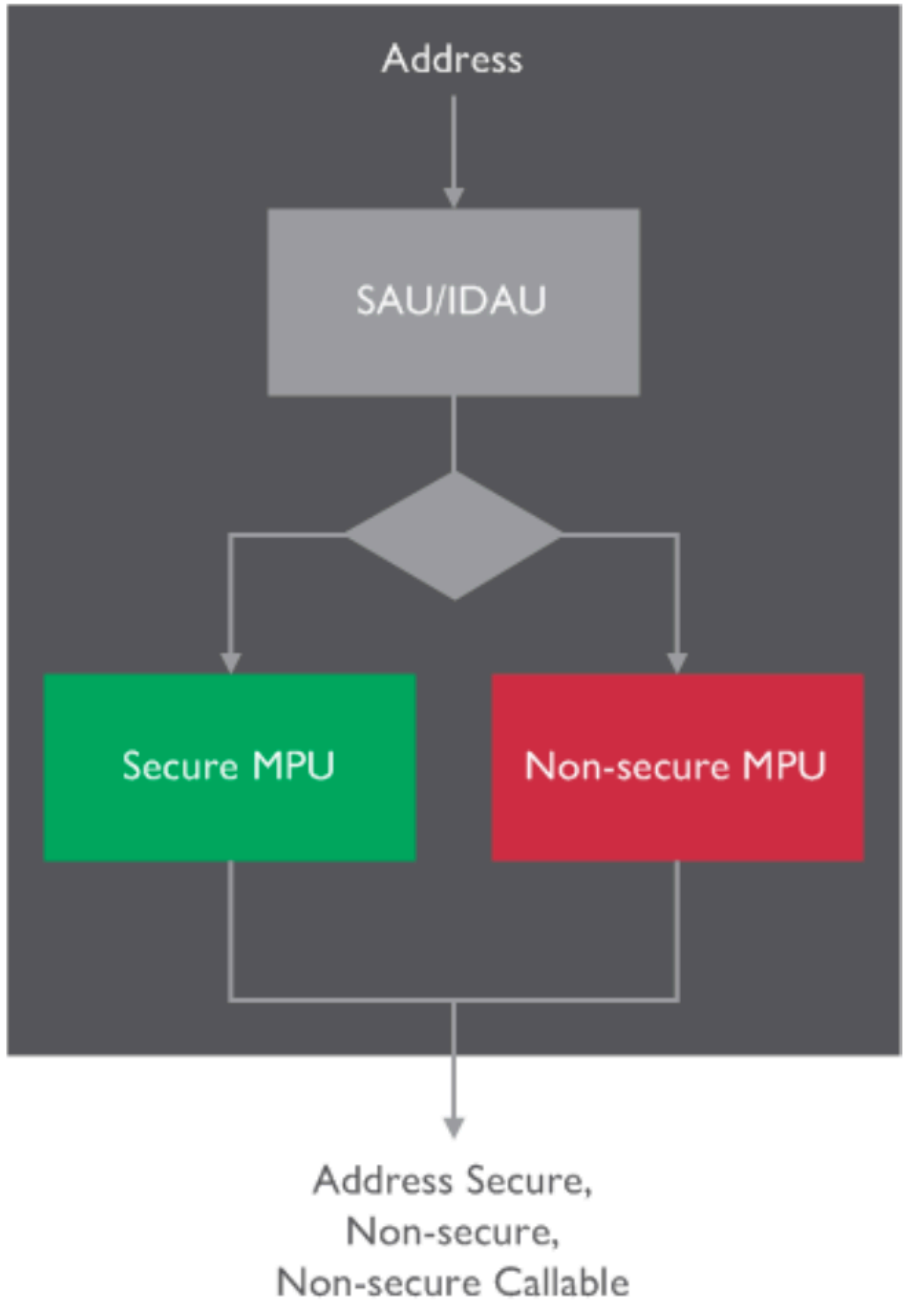


"Software Development in ARMv8-M Architecture" by Joseph Yiu

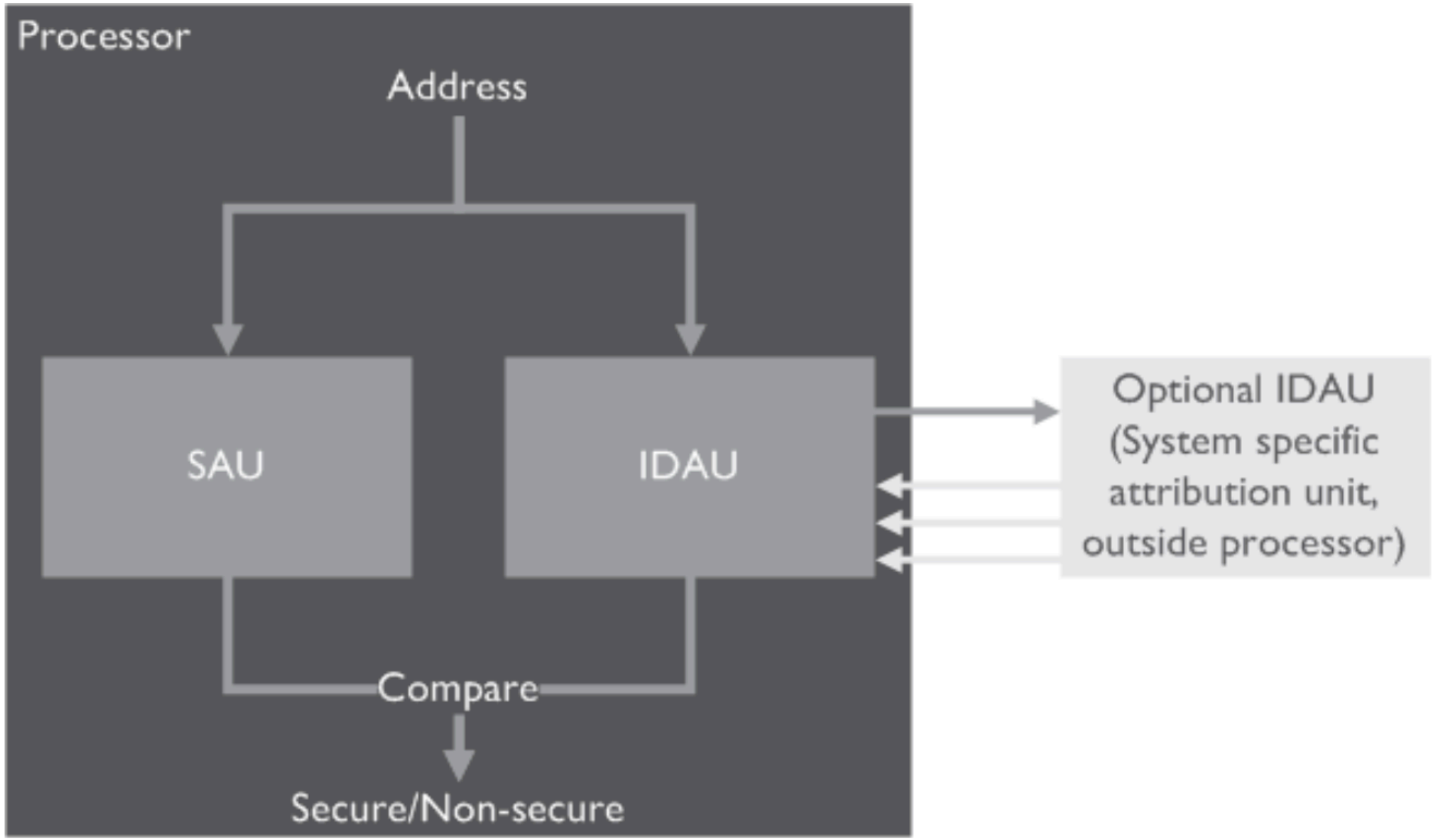

Secure program code
#include <arm_cmse.h>
int __attribute__((cmse_nonsecure_entry))
foo1(int a)
{ // Secure function callable from Non-secure
...
}
int __attribute__((cmse_nonsecure_entry))
foo2(int b)
{ // Secure function callable from Non-secure
...
}
Non-Secure program code
extern int foo1(int a);
extern int foo2(int b);
int main(void)
{
…
foo1(x);
…
foo2(y);
...
}
Compile
Compile
Branch
veneer
address
(linker script)
Link
Export
Library
Link
Secure program image
SG
B.W foo1
SG
B.W foo2
…
foo1:
…
BXNS LR
foo2:
…
BXNS LR
Non-secure program
image
main
…
BL foo1
…
BL foo2



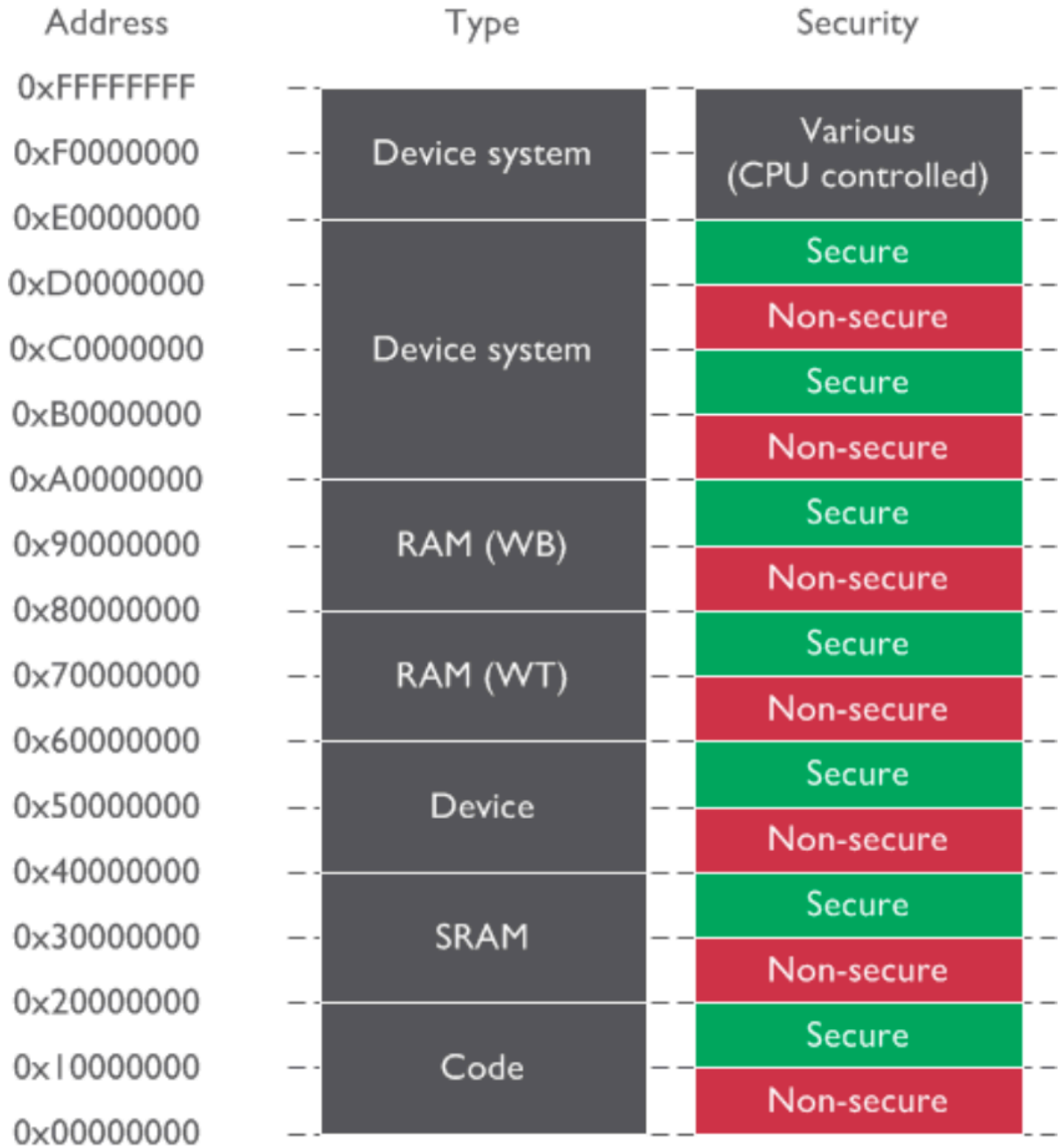
https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf



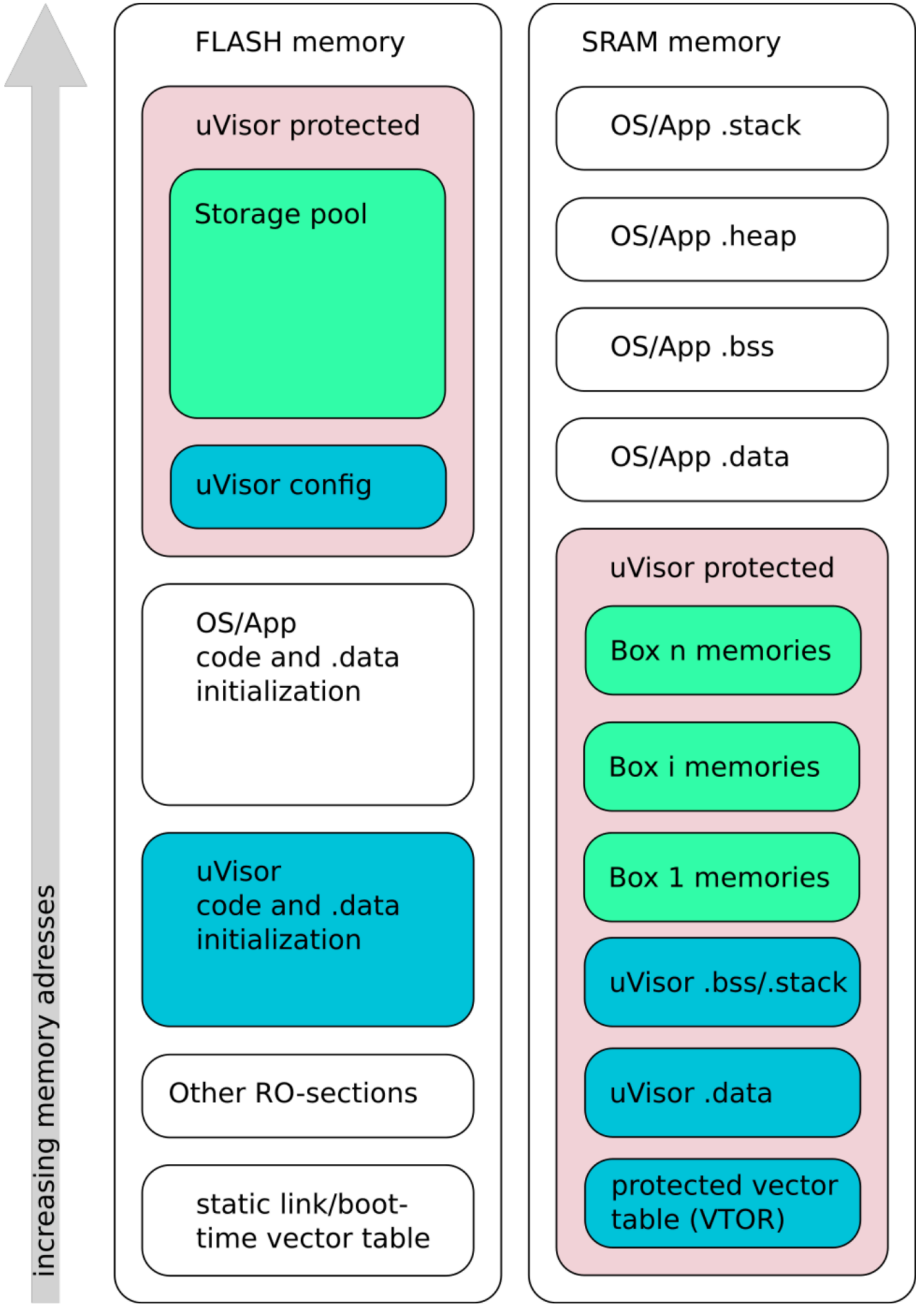
https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

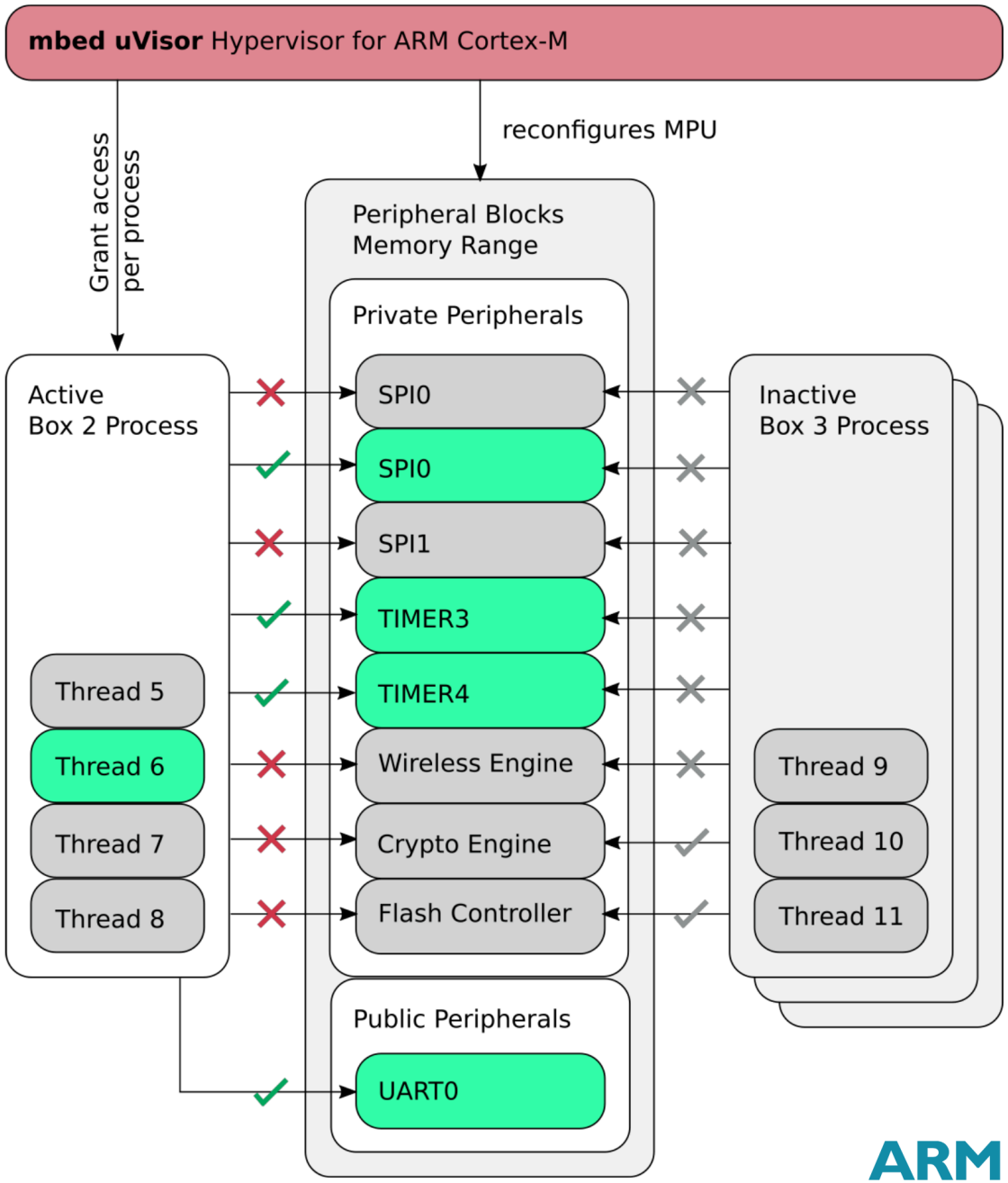


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

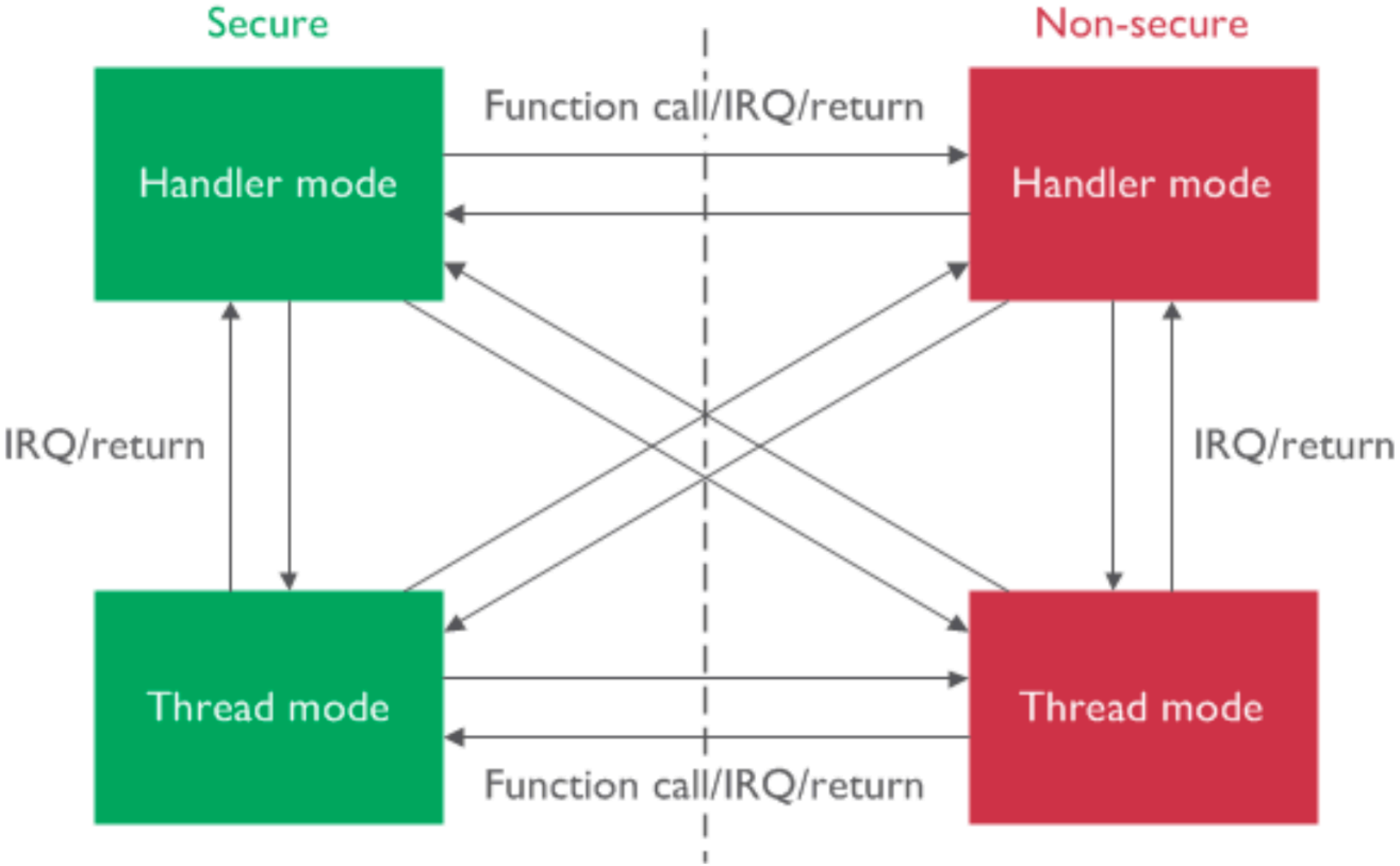


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

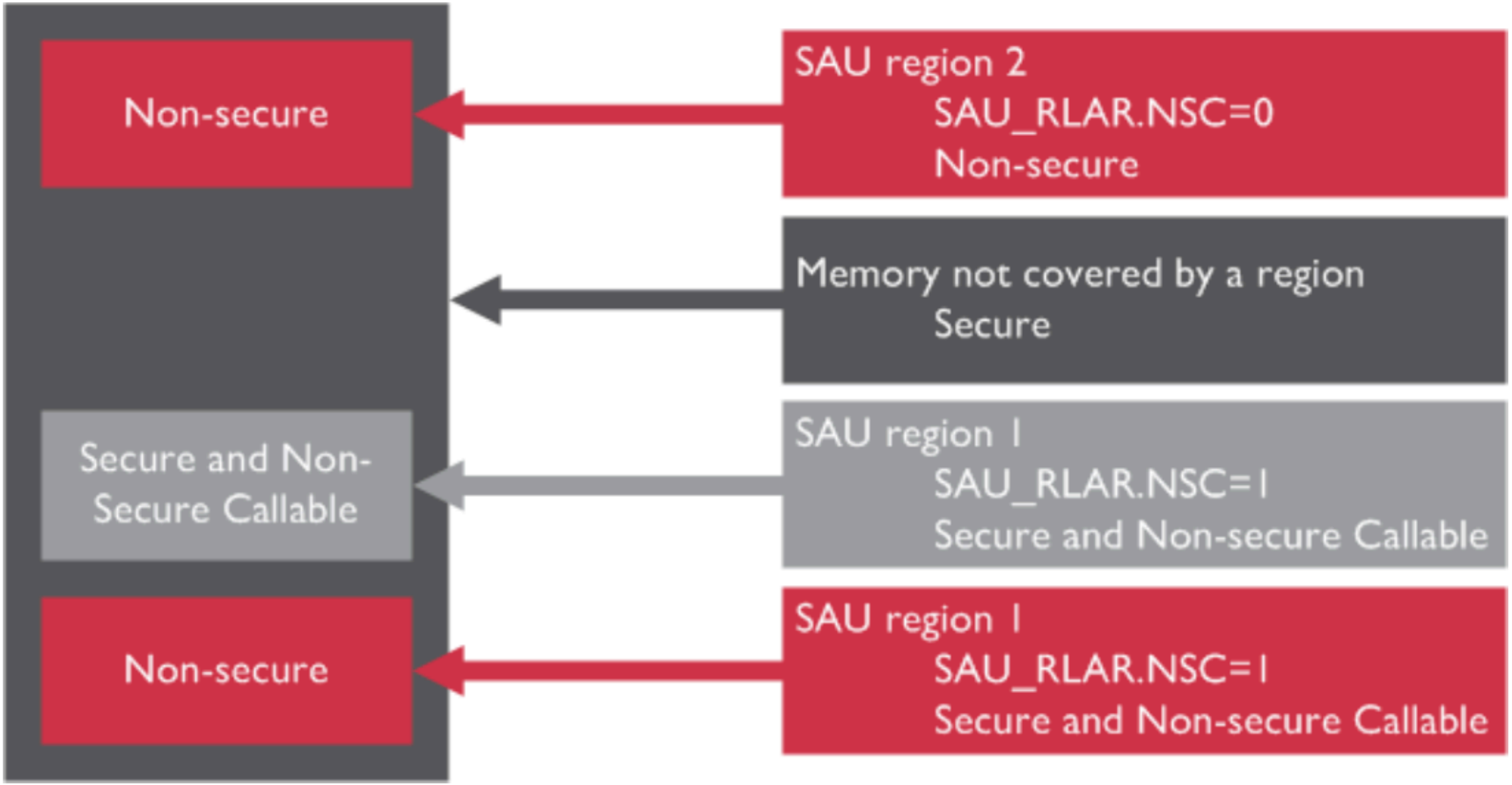




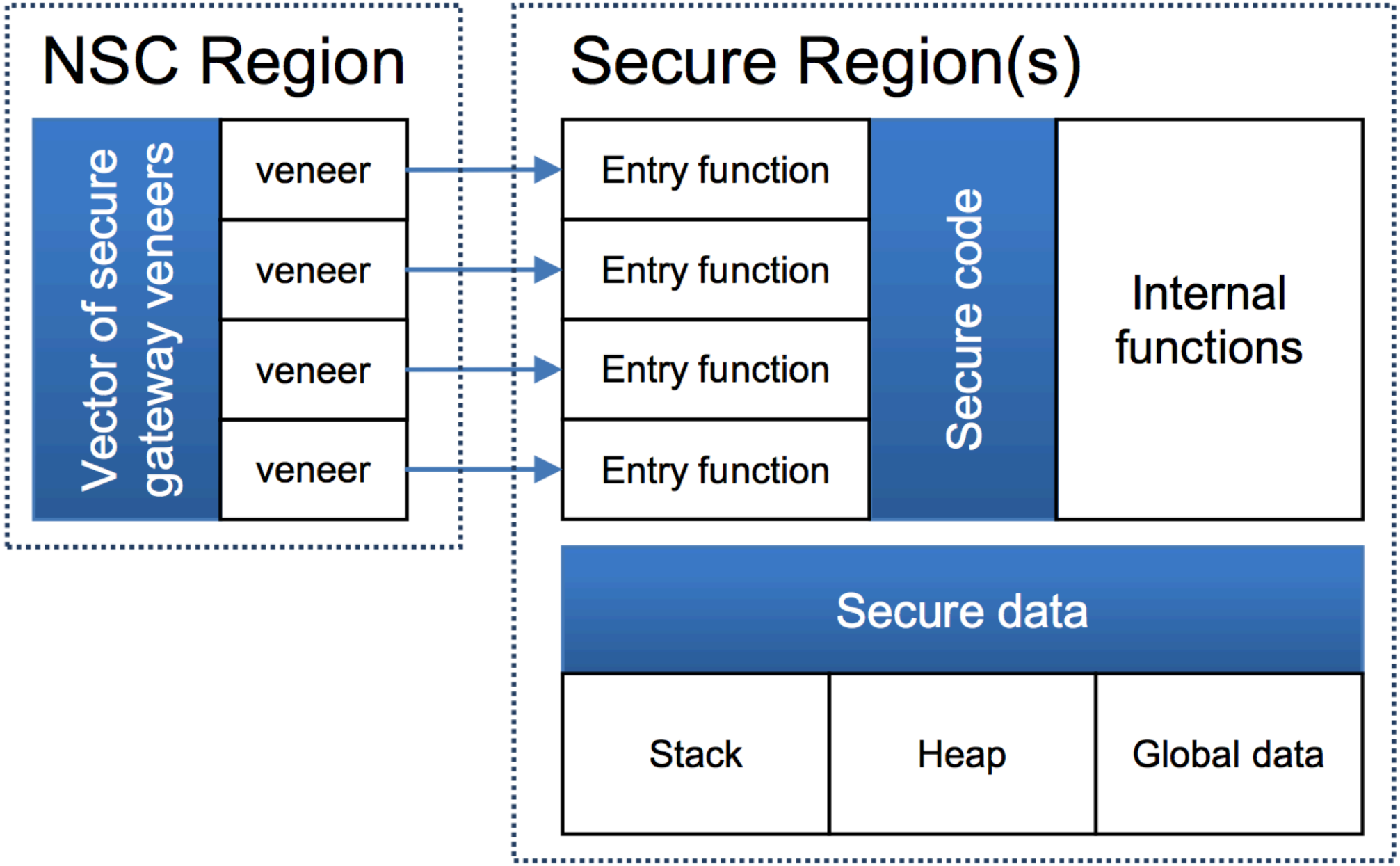
https://github.com/ARMmbed/uvisor/raw/docs/uVisorSecurity-TechCon2016.pdf



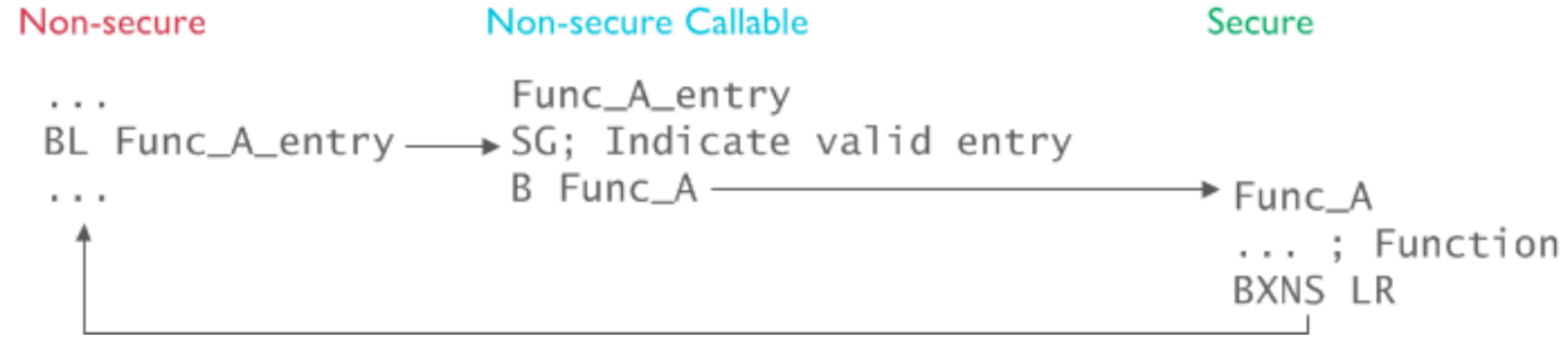
https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

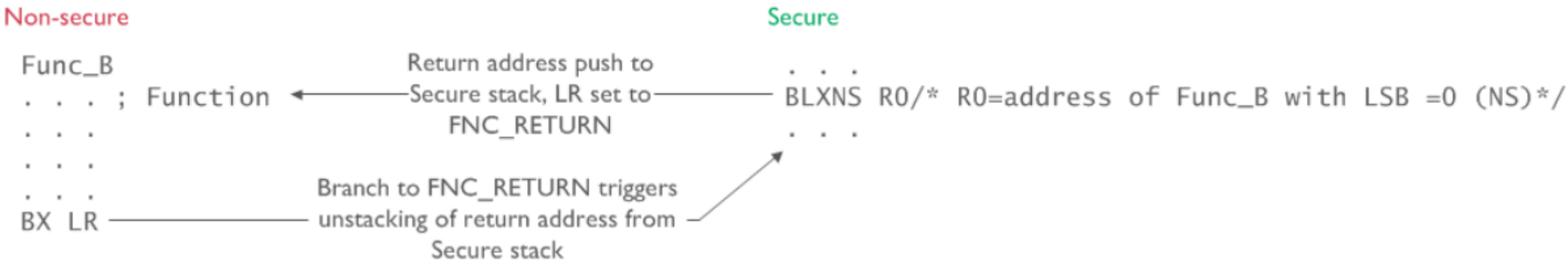


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf

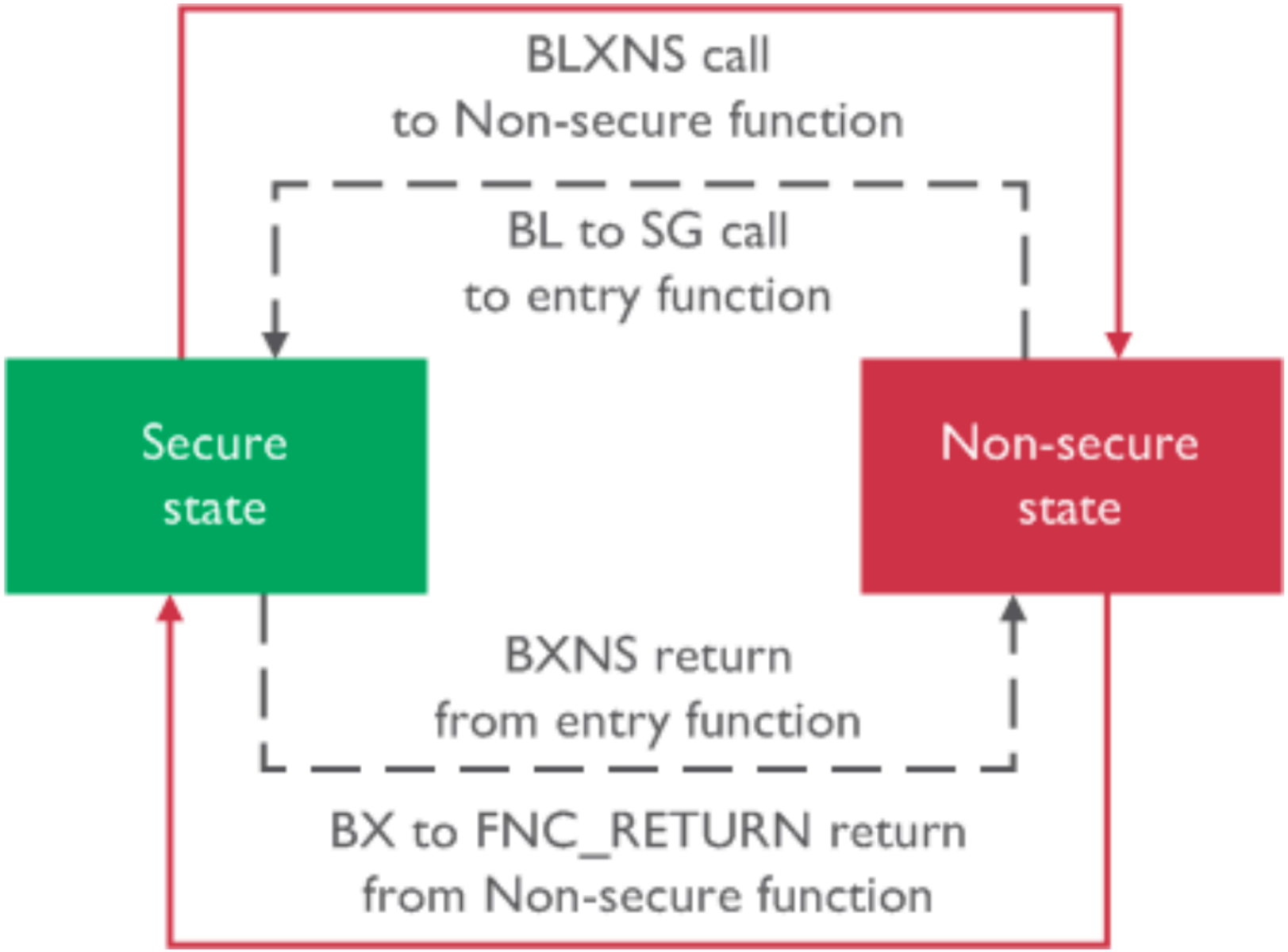


"Software Development in ARMv8-M Architecture" by Joseph Yiu



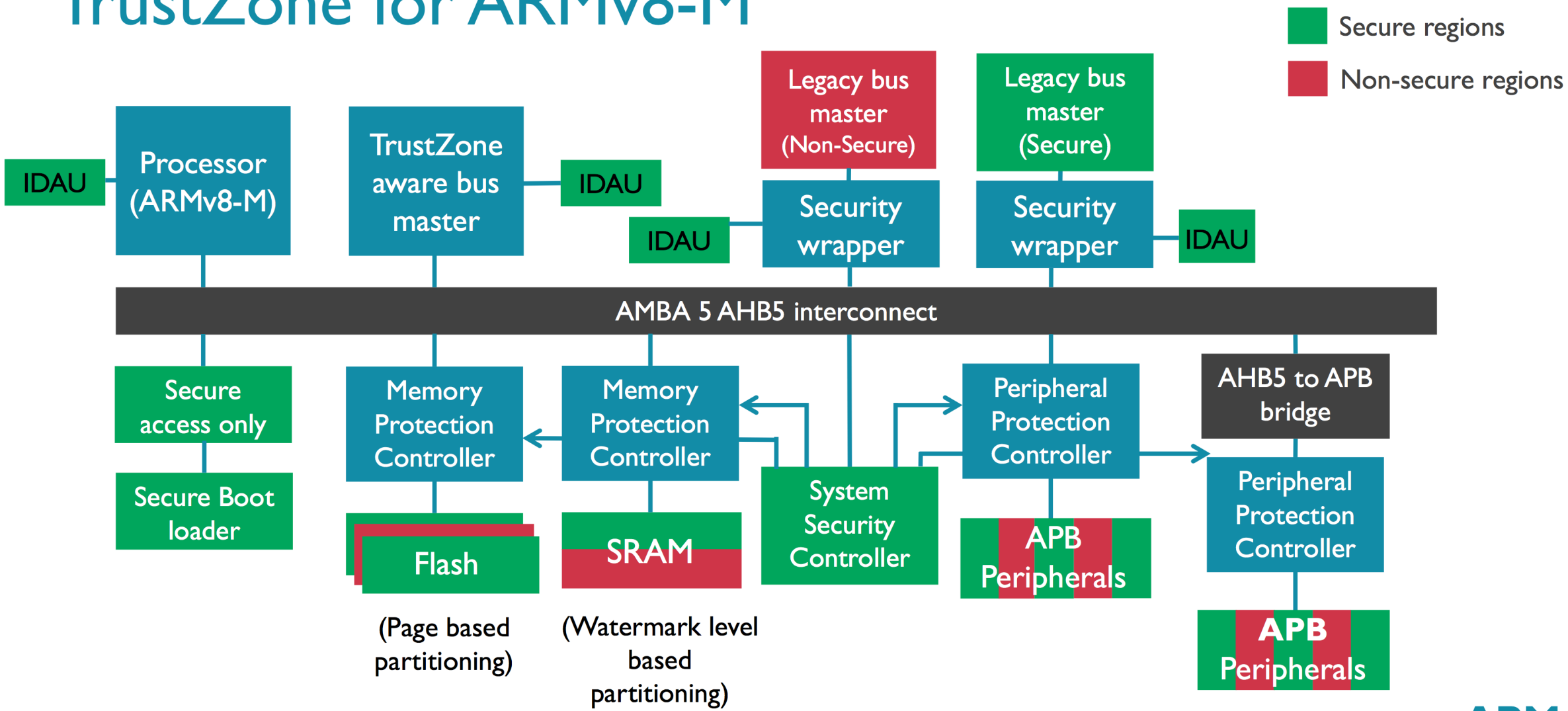


https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf



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

Simplified software security with TrustZone for ARMv8-M



ARM

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

Thank you for listening!

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

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