KU LEUVEN

FACULTY OF ENGINEERING SCIENCE

A Comparative Analysis of Security Features Between Sancus and TI MSP430 IPE

Jolan Hofmans

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, option Secure Software

Thesis supervisors:
Prof. dr. ir. Frank Piessens
Dr. ir. Jo Van Bulck

Assessors:
Dr. ir. Job Noorman
Dr. ir. Alexander van den Berghe

Mentor:
Ir. Márton Bognár

Academic year 2021 – 2022

Preface

Before we jump into the technicalities, I want to take a moment to thank some people.

To kick-off, I want to thank you, the reader, for taking the time to read this thesis.

I would like to thank my supervisor prof. dr. ir. Frank Piessens and co-supervisor dr. ir. Jo Van Bulck to let me work on this topic. A thank you to the entire Sancus working group for their fantastic work to make theses on this topic possible.

A special thanks to my daily mentors Márton Bognár and Jo Van Bulck for their guidance and challenging mindset in introducing me to this exciting but abstract topic. I learned a lot from all the discussions we had.

I want to thank my parents for allowing and giving me the freedom to study in a field I really enjoy.

Jolan Hofmans

Contents

Abstract

Over the last decade, the Internet of Things (IoT) and especially interconnected embedded devices have known a boost. This boost involves an important role in security mechanisms. This is where Trusted Execution Environments (TEE) play an essential role as they bring strong security guarantees. A TEE runs a program in an enclave, this enclave isolates the program from unsecure software and other enclaves.

A TEE often implements specific hardware modules that enforce these security guarantees. These hardware modules are in many cases not present on commercial microcontrollers like MSP430 microcontrollers. However, commercial microcontrollers come with built-in security features like MSP430 Intellectual Property Encapsulation (IPE), which is a security feature that provides isolation properties for a memory segment of code and data. If these security features could guarantee the same strong security guarantees as a TEE, it could be possible to create a custom TEE with an off-the-shelf microcontroller to lower the cost.

This master's thesis aims to compare the Texas Instruments (TI) MSP430 microcontrollers with Sancus to conclude whether MSP430 can provide the same security guarantees as Sancus. Sancus is a security architecture that provides isolation for multiple enclaves. A software provider can verify if the uploaded software that is installed is unmodified, this is remote attestation. Such a mechanism often uses cryptographic primitives. For this comparison, we focus in particular on MSP430 IPE as this could provide isolation properties for an enclave.

Before starting the comparison, we first construct security objectives and an attacker model for MSP430 IPE. A comparative analysis of MSP430 and Sancus was made based on three properties. The first two properties are isolation and attestation. The third property is Direct Memory Access (DMA). A novel DMA attack was discovered on Sancus, and the DMA capabilities on both MSP430 and Sancus were compared based on the hardware design to check if the attack would also be possible on MSP430.

The comparison of the isolation mechanism showed some security limitations in MSP430 IPE. These limitations are mainly hardware limitations that can be exploited in several methods. Software limitations give an attacker more tools to exploit the hardware limitations. Countermeasures are proposed throughout the discussion of security limitations. These security limitations are combined to zero out the key of the example code TI provides on IPE. In another attack, these security features are used to extract the key from the protected IPE section in the example code.

Samenvatting

Gedurende het laatste decennium hebben het Internet of Things (IoT) en in het bijzonder embedded devices een opmars gekend en zijn nu niet meer weg te denken uit ons alledaags leven. Ze zijn onder andere van toepassing in systemen die instaan voor onze veiligheid of systemen die gevoelige informatie opslaan. Dit is waar Trusted Execution Environments (TEE) een belangrijke rol spelen aangezien ze sterke beveiligingsgaranties met zich meebrengen. Een TEE voert een programma uit in een zogenaamde enclave, deze enclave isoleert het programma van andere enclaves en onbeschermde software.

Een TEE implementeert vaak specifieke hardwaremodules die deze beveiligingsgaranties garanderen. Deze hardwaremodules zijn in vele gevallen niet aanwezig in commerciële microcontrollers zoals bijvoorbeeld MSP430 microcontrollers. Deze commerciële microcontrollers hebben ingebouwde beveiligingsmodules zoals MSP430 Intellectual Property Encapsulation (IPE), dit is een beveiligingsmodule die isolatie eigenschappen voorziet voor een segment in het geheugen van de microcontroller dat code en data opslaat. Als deze beveiligingsmodules dezelfde beveiligingsgarantie kunnen geven als een TEE, dan is het mogelijk om een op maat gemaakte TEE te ontwerpen met een reeds beschikbare microcontroller om de kost te verlagen.

Deze masterproef vergelijkt (Texas Instruments) TI MSP430 microcontrollers en Sancus om te beoordelen of MSP430 dezelfde veiligheidsgaranties kan bieden als Sancus. Sancus is een beveilingsarchitectuur die meerdere enclaves van elkaar isoleert. Verder kan een softwareprovider controleren of de geïnstalleerde code onveranderd is gebleven, dit is attestatie. Dit soort mechanisme maakt vaak gebruik van cryptografische primitieven. Voor deze vergelijking wordt vooral gefocust op MSP430 IPE aangezien dit isolatie eigenschappen voor een enclave zou kunnen bieden.

Alvorens te starten met de vergelijking, stellen we eerst de beveilingsdoelstelling en een bedreigingsmodel op MSP430 IPE. Een vergelijking van MSP430 en Sancus wordt geanalyseerd op basis van drie eigenschappen. De eerste twee zijn isolatie end attestatie. De derde eigenschap is Direct Memory Access (DMA). Een nieuwe DMA-aanval werd ontdekt op Sancus en de DMA-mogelijkheden van zowel MSP430 en Sancus worden vergeleken gebaseerd op het hardware-ontwerp om na te gaan of een gelijkaardige aanval ook effect zou hebben op MSP430.

De vergelijking van de isolatiemechanismen toonde enkele beveiligingslimitaties in MSP430 IPE aan. Deze limitaties zijn hoofdzakelijk hardware limitaties die op verschillende manieren uitgebuit kunnen worden. Softwarelimitaties geven een

aanvaller meer hulpmiddelen om de hardwarelimitaties uit te buiten. Maatregelen worden voorgesteld doorheen deze hele discussie over deze limitaties. Deze beveiligingslimitaties worden gecombineerd om sleutel in de voorbeeld code van TI op nul te zetten. In een andere aanval worden deze beveiligingslimitaties gebruikt om de sleutel uit de beveiligde IPE-sectie te halen in de voorbeeld code.

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

BSL	Bootstrap Loader
CPU	Central Processing Unit
DMA	Direct Memory Access
DRoT	Dynamic Root of Trust
FRAM	Ferroelectric Random-Access Memory
IoT	Internet of Things
IPE	Intellectual Property Encapsulation
IVT	Interrupt Vector Table
MCU	Microcontroller Unit
ROP	Return-Oriented Programming
RoT	Root of Trust
SIA	Secure Intermittent Architecture
TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TI	Texas Instruments

Chapter 1

Introduction

In the last decades, computing devices have become a dominant part of our lifestyle as we rely on them for everyday activities. As the computing power increased, these devices evolved from room-filling machines to smartphones and microcontrollers. However, all of these machines are designed to run several applications and can, in some cases, be used by several users. These applications do not need to trust each other. To run them isolated from each other different solutions were designed from the hardware level to the operating system. In high-end devices, one could rely on a large trusted computing base (TCB), a set of components that can be trusted to perform the security guarantees. This set is complete to guarantee the security objectives. It could even include the entire operating system.

The Internet of Things (IoT) has known an immense growth in recent years, with billions of interconnected embedded devices. Nowadays, these devices stand in for real-world safety applications or handle sensitive data. The necessity for similar security guarantees as their high-end counterparts has also grown. However, implementing IoT devices brings different challenges in contrast to high-end devices. IoT devices come with power constraints, limited memory space, and computing power and space restrictions as they need to be as small as possible.

Trusted Execution Environment Trusted Execution Environments (TEE) were the result of the research for these isolation mechanisms on embedded devices [29, 34]. There are commercial results like ARM TrustZone [39] as well as academic solutions like Sancus [33], VRASED [35], and Keystone [27].

The idea of a TEE is to run a program inside an enclave or module. This enclave protects the program from other programs and processes running on the same device. Furthermore, a software provider wants guarantees that the software that they uploaded is in the same state as they intended - meaning unmodified. This is another concern for a TEE to provide a mechanism that a software provider can check this. A TEE comes with specialized hardware modules that provide these security guarantees. These hardware modules are necessary as a TEE does not trust the underlying operating system. A TEE aims for a minimal TCB because of all the limitations an embedded device brings with it. Therefore the set of these hardware

modules needs to be as small as possible.

Commercial Microcontrollers For most of these TEEs, hardware modifications are needed to expand the chip with some essential hardware modules that need to support the TEE. When looking at commercial microcontrollers, there are general-purpose microcontrollers that come with an already built-in set of hardware components. When these components could give the same security guarantees as a TEE, this would be an interesting path to think about as one could take an off-the-shelf microcontroller and use the onboard features to create a custom TEE. This thesis looks at the TI MSP430 series and whether these microcontrollers could create their custom TEE with built-in security features. In this thesis, we focus mainly on MSP430 Intellectual Property Encapsulation (IPE) [37] as this provides isolation properties for code and data similar to the isolation mechanism necessary for an enclave. In this analysis, we will compare MSP430 IPE with Sancus. Sancus is a security architecture that provides isolation for multiple enclaves and allows software providers to attest their uploaded enclaves. As Sancus is implemented on OpenMSP430 [7], an open-source version of MSP430, Sancus and MSP430 IPE have common base for this comparison.

Need for Security These embedded devices are connected to the network, making them vulnerable to various attacks. Furthermore, an attacker has strong capabilities [38], which allows them to upload code to and run code on an embedded device. When code is not properly protected, this creates possibilities for an ROP attacker. Another way of attacking these devices is through side-channels, microarchitectural details that leak some information to the attacker.

As for Sancus, the only two known attacks that succeed are Nemesis [49], and the novel DMA attack [1] because of its strong security guarantees. This is an interesting choice to compare with MSP430.

1.1 Contributions

The goal of this master's thesis is to compare the academic TEE Sancus with the commercial TI MSP430 microcontrollers, specifically with the IPE isolation mechanism.
This thesis has the following contributions:

- A summary of the security objectives of MSP430 IPE. As TI stated the security objectives not in a transparent manner, we combined multiple quotes and discussed them to come up with some clearly stated security objectives. Furthermore, we provide an attacker model for MSP430 IPE.
- An in-depth comparison of the isolation and attestation mechanisms from Sancus and MSP430 IPE. We highlight differences in the design decisions they made on the hardware and the software level and remark on which potential consequences these decisions could have.

- A comparison of the DMA capabilities used in Sancus and MSP430 in terms of hardware design, purpose, and accessibility. This comparison discusses the attacker surface an attacker can cover with DMA capabilities on Sancus and MSP430.
- Experimental results on security limitations that were discovered on MSP430 during the comparison of Sancus and MSP430 IPE were exploited to extract the key of the example program TI provides.

1.2 Outline

Chapter 2: In this chapter an overview is provided of trusted execution environments. After this, a more detailed explanation is given about Sancus. This is followed by an overview of the MSP430 microcontroller series. Next, an explanation of relevant side-channel attacks on Sancus is given. The last part gives a short instruction on return-oriented program attacks.

Chapter 3: This chapter compares Sancus with the security features present in MSP430. First, we describe and summarize the security objectives and attacker model from MSP430 and compare them with those from Sancus.

Next, we discuss the concepts of isolation and attestation by pointing out differences concerning design decisions and some exciting alternatives or improvements found in the literature.

In the last discussion, we point out the differences in DMA capabilities between MSP430 and Sancus and give a reasoning on how a side-channel DMA attack relates to MSP4340.

Chapter 4: The comparison shows some security limitations in the TI MSP430 IPE design. This chapter defines three attack primitives and demonstrates how they can be exploited with some straightforward examples.

We combined these attacker primitives to show experimentally that the example code MSP430 provides is not secure by demonstrating two different attacks.

Chapter 5: In this last chapter, an overview of the conclusion is provided. The contributions are summarized. Furthermore, we highlight the limitations of this work and give some indications for future work.

Chapter 2

Background

2.1 Trusted Execution Environment

Trusted Execution Environments were created as there was a need for trusted computing on embedded devices [29, 34]. There is no unified definition of TEE. Sabt et al. [41] attempted to merge all definitions into one definition. Maene et al. [29] propose a list of security features that a TEE must suffice. From this definition arise different properties a TEE needs to satisfy:

Isolation Code and data from an enclave need to be protected from interaction with other processes running on the MCU. This interaction can be interpreted in different manners. Another process cannot execute random snippets from the enclave. If this were the case, the enclave would be vulnerable to Return-Oriented Program (ROP) attacks. A straightforward way to implement isolation is a program counter-based hardware component [44] that checks if a process has access to a memory area. This check is based on the location of the program counter. The protected area is only accessible if the program counter is located inside a predefined region.

Secure Storage Secure storage means that the stored data is confidential, meaning no third party can read it. Another aspect is the integrity of the data, meaning the data is not modified. The last aspect is the freshness of the data to prevent replay attacks where an attacker stores a previous version of the code inside the memory. Sealing the storage is a typical manner to reach these goals. Sealing means that only the enclave can check the state of the data with a cryptographic method where only the enclave has access to the key.

Attestation A software provider wants guarantees that the software they installed or updated is in the state they intend it to be. This means there is no modification between the moment the software provider did send the image - binary representation of the code - and the moment isolation takes over the protection after the code is stored in memory. Remote attestation gives a trustworthy solution as this allows the software providers to challenge the stored enclave to prove its authenticity.

Root of Trust (RoT) Trust means that a process or a component does what it is expected to do. There are two different forms of trust. Static trust means that trust is only checked before the system is deployed. Dynamic trust measures the trust during the entire lifecycle of the system. A TEE needs to guarantee trust during this entire life cycle. TEEs provide a Dynamic Root of Trust (DRoT). This DRoT can be seen as a separate hardware module that measures the trustworthiness of the components every time the microcontroller is rebooted. These components form a trust chain that allows the protection mechanism to verify that they will work as they were intended to do.

ARM TrustZone TrustZone [39] is a security extension for ARM processors. It partitions the hardware and software into two separate zones called worlds, a normal world and a secure world [31]. The normal world contains all non-secure software, while the secure world is the enclaves. A hardware barrier assures that components from the normal world cannot access the secure world.

2.2 Sancus

Sancus [33] is a security architecture targeting embedded systems connected to a network where multiple software providers can deploy and update enclaves on an embedded device. Sancus has application in smart electricity meters [30], and it is leveraged for efficient message authentication in vehicles [48]. Sancus provides the following properties:

Isolation Sancus provides guarantees that software modules are protected from each other. A software module's data section can only be accessed if the program counter originates from the text section of the same software module. Text sections can only be accessed from their entry points. Writing to a text section is not allowed. A text section can only be executed when the program counter is located in the text section itself. These rules result in table 2.1.

Remote Attestation While isolation protects when its state is stored in memory. A software provider wants to ensure that the code uploaded to and stored on the board is unmodified. Once the software module is stored in memory, the software provider will send a nonce. The software module uses an encrypt instruction on this nonce the get the key, and it will send the MAC, a part of the key, to the software provider. The software provider can now check if the key is correct, and thus the software module is unmodified.

Secure Communication A software provider can communicate with a software module on a node through a protected communication line. Two software modules on the same node can communicate with each other without another software model eavesdropping or tampering with the communication.

From/To	Entry	Text	data	Unprotected
Entry	r-x	r-x	rw-	rwx
Text	r-x	r-x	rw-	rwx
Unprotected/Other SM	- -x	- - -	- - -	rwx

TABLE 2.1: Memory access control from Sancus 2.0. Table from [33]

Software Module Sancus runs one or more software modules. A software module consists of a text section and a data section. A text section stores code and constants, it has an entry point to access the section from outside the software module. A data section stores runtime data like, for example, the stack and variables.

A protected storage area keeps track of all software modules on the MCU. Figure 2.1 visualizes this management. For each software module, it stores the start and end address of the text and data section together with the private key of the module and an ID. The addresses of the section are important to enforce the isolation. The stored key is computed based on the content of the text section of the software module and the start and end address of the text and data section. This key is used to attest the state of the software module.

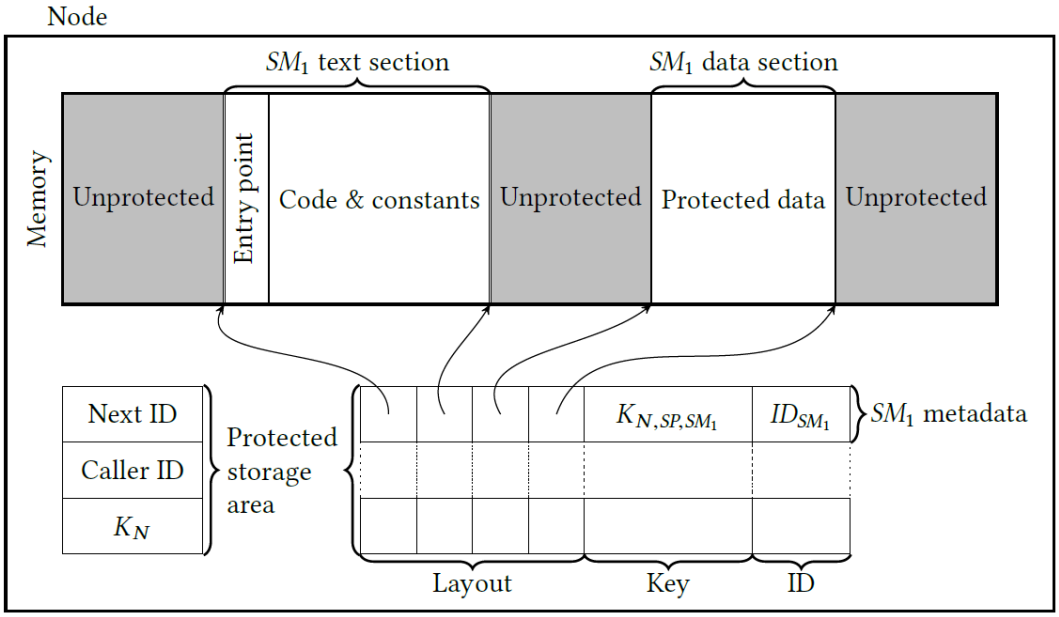

FIGURE 2.1: Representation of a software module stored in the memory of a node, and Sancus manages these software modules. Figure from [33]

.

DMA Sancus is built on the OpenMSP430 architecture [7], the open-source version of the MSP430 architecture. Sancus 2.0 [33] does not yet support DMA. Seminara

designed a DMA controller [43] for Sancus, and later on, Sancus was extended with the DMA interface of OpenMSP430.

OpenMSP430 has three memory partitions: program memory, data memory, and peripheral memory. Figure 2.2 represents how OpenMSP430 interacts these memory partitions. Those three partitions can be accessed in parallel, and a transfer takes one cycle for one byte. The CPU and DMA interface can request to access the same memory partition at the same time, but only one can access the memory partition at a time. Therefore the CPU has prioritized access over the DMA interface.

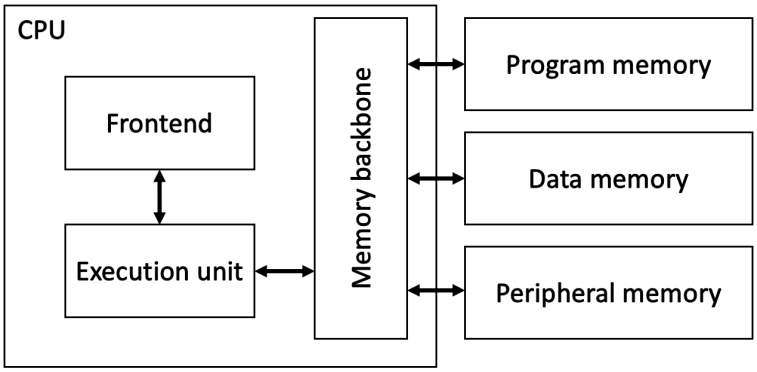

Figure 2.2: Representation of memory interaction on OpenMSP430 based on [7, 33].

.

Attacker Model Sancus considers attackers with strong capabilities. An attacker can modify all software from the MCU. The operations system on the board can also not be trusted as the code of the operating system does not be considered secure. From the perspective of a software module, it can trust itself, Sancus, and other hardware-like peripherals, as shown in figure 2.3.

An attacker can control the communication line between a software provider and the MCU. He can put himself in a man-in-the-middle position where he can sniff all the traffic or even tamper with the packages.

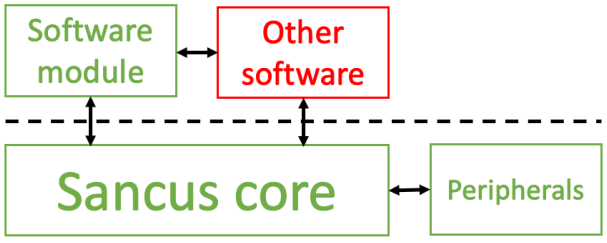

Figure 2.3: Attacker model of Sancus. Simplified attacker model from [2]

.

2.3 MSP430

MSP430 is a series of 16-bit microcontrollers[11]. The architecture is a von Neumann architecture where the program and data are stored in the same memory [50]. MSP430 MCUs are known for their low cost, high speed, and low power consumption. It is a complete system-on-a-chip, meaning several components are provided inside the chip, like memory, RAM, timers, and analog to digital converter. The MSP430 series is used for general purposes and metering applications like power controlling [23] and standing on for secure communication of portable devices for medical and fitness purposes [42].

MSP430FR5969 MSP430FR5969 combines the properties of the main memory FRAM and a low power system architecture to provide a 16-bit MSP430 CPU [15].

Figure A.1 shows a schematic of the functional blocks present on the MSP430FR5969 chip. Remark the presence of peripherals like a DMA controller, non-volatile memory in the form of FRAM, and volatile memory as RAM. Furthermore, MSP430FR5969 has a peripheral that executes cryptographic methods called AES256. These components will be explained in the following paragraphs.

The MSP430FR5969 chip is mounted on the MSP-EXP430FR5969 launchpad [20]. Figure A.2 shows the layout of the launchpad. We provide an overview of the more non-trivial components of the launchpad:

Emulation MCU and Energy Trace Technology The Emulation MCU is an on-board debugger to easily connect to a computer for a smoother program and debug experience. The Energy Trace Technology is an analysis tool that is useful for power measurements of various components, which can be used when debugging to minimize power consumption.

Crystals Clock applications use the crystals, or resonators, to create specific frequencies as they help calibrate the frequency.

Super Cap The Super Capacitor is a built-in battery that can be used as a stand-alone power supply while the launchpad is not connected to an external power supply.

Memory Map The MSP430 FR-series contains a ferroelectric random-access memory (FRAM) as the main memory. Furthermore, an extra ROM for the BSL and random-access memory (RAM) are provided.

FRAM has the read and write advantages of RAM while it is non-volatile - it preserves its state when the power is down - and consumes less power than other non-volatile memory alternatives, and has no limit on the amount of writes [13].

2.3.1 DMA Controller

MSP430 contains a DMA module [17, 15, 14] to minimize its power consumption. Besides this, DMA reduces the load on the CPU and increases the throughput of

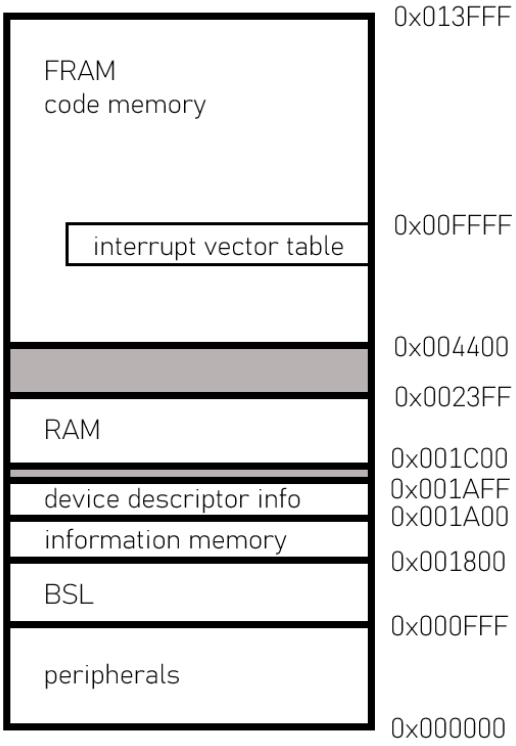

Figure 2.4: Memory map of the TI MSP430 [15, 14].

data from peripherals. A DMA controller can have up to 8 channels (FR5969 has 3), where each channel has one of four addressing modes and one of six transfer modes. The DMA controller can only handle one transfer at a time. Therefore the channels need to be prioritized, which can be done in two ways. The first way, which is default, is to prioritize channels by giving each channel a fixed priority, where the highest priority transfer is handled first. A second manner dynamically assigns priorities by how long ago the channel has made a transfer by setting the ROUNDROBIN bit. How longer since the last transfer, the higher the priority of the channel. A trigger is needed to initiate a transfer. This trigger can be set independently for each channel in the DMAxTSEL register. For each transfer, one or two cycles are needed to synchronize with the clock signal of the board. Afterwards, for each byte that will be transferred, an additional two cycles are needed. At the end of the transfer, an extra cycle is needed to finish the transfer.

Transfer Modes

Single Transfer DMA controller needs a separate trigger for each byte or word that needs to be transferred. DMAxSZ contains the number of transfers that need to be executed. DMAxSA and DMAxDA contain respectively the source and destination address for a transfer, and DMASRCINCR and DMADSTINCR need to increase or decrease these addresses every transfer.

Block Transfer During a block transfer mode, the DMA controller transfers a block of data while only one trigger is needed. DMAxSZ determines the size of the block.

Burst-Block Transfer This mode is similar to a block transfer, except for the fact that there is an alternation between transferring data and CPU execution time. After transferring four bytes of the block, equivalent to consuming eight cycles, the CPU gets two cycles to execute.

Repeated Modes Repeated single transfer, repeated block transfer, and repeated burst-block transfer differ from the non-repeated modes in the sense that the DMAEN stays enabled after the transfer is done, resulting in the start of the next transfer.

2.3.2 Security Features

Several security features are implemented in the MSP430FR5969. This section summarizes these features.

Bootstrap Loader

The Bootstrap Loader (BSL) [19] has a variety of applications: in the field firmware updates, program RAM and FRAM using a UART or I2C interface, and read memory. Commands are used to execute these applications, except for three commands. All these commands are protected by a password. The three unprotected commands can provide the password to get access to the BSL, erase the entire memory, and change the baud rate for the following packages. The password is 32 bytes long and is based on the end of the interrupt vector table. Every time a new program is loaded onto the device, a new password is set. The more interrupts are used in the program, the more complex the password will be. At compile time, the empty locations in the interrupt vector table are filled with the trap interrupt. When the user enters a wrong password, a mass erase will be executed, which will erase all memory except for the IPE and the IPE settings. Afterwards, the device can be accessed again with the default password. A program cannot access just any part of the BSL. Only 4 bytes are accessible, called the Z-area.

IP Encapsulation

IP Encapsulation [37, 14, 24] protects a specific part of the FRAM chosen by the user. It prevents read and write access from outside the area to the inside of the area. The area can only be removed with a special mass erase. Interrupts raised from outside the area can be disabled but this is not default. Registers from the CPU and peripherals are not cleared automatically. The programmer is responsible for doing this manually.

An IPE segment contains several sections. At least the first two are always present. The other sections are only present when explicitly stated:

struct-section contains data for initialization and address location of the other sections.

function-section is the place where functions belong.

cons-section contains constant data.

isr-section contains the interrupt service routines used by the code in the IPE-segment.

vars-section contains variable data used by the code in the IPE-segment.

The memory range that belongs to the IPE is stated in two registers, defining the lower border (IBL) and the higher border (IBH) of the IPE section. These two registers are used to determine whether a certain memory address can be accessed. Once they are set, they cannot be modified. Only a special erase function from the BSL can clear them. JTAG or DMA will never be able to access the IPE segment. The CPU can only access the IPE segment if the program counter is located inside the IPE segment. Suppose the CPU reads a word from the IPE segment, the read value will be 0x3FFF. The IPE segment can be initialized before the execution of the application is started.

Segmentation

The MPU [14] has the ability to divide the memory into three partitions where every partition is at least 1 kB. Each partition can be given read, write, or execution permission. These permissions are fixed and can be changed when the device gets a program update or when the correct password is provided at runtime.

Segmentation and IPE can interoperate with each other. An IP encapsulation can overlap with two segments from the segmentation, meaning each partition of IP encapsulation inherits the permission from the partition it belongs to.

AES Accelerator

MSP430 has a module called AES accelerator [14] which performs encryption and decryption of 128, 192, or 256 bits data with respectively a key of 128, 192, or 256-bit length. The amount of cycles necessary to perform an AES operation is fixed depending on whether it is an encryption or decryption and the size of the key. Tables 2.2 and 2.3 gives more details about the amount of cycles needed for which operation. The AES accelerator leverages DMA operation for access to keys and data blocks.

The accelerator can be configured to perform an encryption or decryption and the length of the input by control registers. The registers that are used to input the key and data count the amounts of words/bytes. When this reaches the amount specified in the control register, the accelerator starts calculating the challenge.

Key	Cycles
128	168
192	204
256	234

TABLE 2.2: Number of cycles needed for an encryption based on the key length. [14]

Key	Cycles
128	168
192	206
256	234

TABLE 2.3: Number of cycles needed for a decryption based on the key length.[14]

Signature 1	Signature 2	Device Security
5555h	5555h	locked without password
AAAAh	length of password	locked with password
any other value	any other value	unlocked

TABLE 2.4: Signature modes of JTAG [37].

Crypto-Bootloader

The crypto-bootloader [8] is a customized bootloader. It provides a more secure communication line between the software provider and MCU with cryptographic primitives. It is used for more secure in-the-field software updates as these cryptographic primitives authenticate the software provider, keep the transferred code confidential, check if all packages of the transferred code have arrived, and secure the integrity of the packages.

When a device reset happens, the crypto-bootloader starts executing and decides whether it stays in bootloader mode or starts the execution of a user program. Crypto-Bootloader must be the only way to access the device, meaning JTAG is disabled and Crypto-Bootloader is preferred over the MSP BSL. By using both, MPU and IPE, the Crypto-Bootloader code will be secured from external access. Furthermore, UART and I2C are the only two options for bootloader communication. They cannot be used both at the same time.

JTAG

TI provides a JTAG [37] interface on the MSP430 series, which programmers can use for debugging, testing, and programming the device. When the device enters its operational lifetime, the owner of the device wants to ceil this access point. Two versions of JTAG are used in the MSP430 series. The first version is a JTAG interface without a password where the user can lock the JTAG, and access can only be regained through the BSL. The second version is a JTAG interface with a password that can be configured based on two signatures. Different configuration can be seen in table 2.4. When JTAG is locked with a password, the length of the password should be chosen with caution. A long password can overlap with the interrupt vector table, meaning the overlapping part should have the correct values from the interrupt vector table.

2.3.3 Instruction Set

The MSP430 instruction set is a RISC instruction set[11]. This means that it is a specialized instruction set optimized for efficient calculations and several addressing modes in order to fulfill the real-time requirements. The MSP430 instruction set [14] contains 27 instructions. They have a length of one, two, or three words. Each instruction contains one word that contains the opcode of the instruction and the addressing mode for the arguments of the instruction. For each argument that has an addressing mode that does not involve the general-purpose registers of the CPU, an extra word is added to the instruction is added. If an argument involves a general-purpose register, this is directly included in the first word of the instruction. Instructions can be grouped in three partitions:

Double-Operand Instructions Double-operand instructions consist of three words. The first word contains the opcode, type of the source, and type of the destination. The next two words contain the source and destination.

Single-Operand Instructions Single-operand instructions consist of two words. The first word contains the opcode and the destination. The next word contains the destination.

Non-Operand Instructions These instructions consist of one word. The first class of these instructions is jump instructions. These consist of an opcode followed by a condition - the condition on which will be decided to jump or not - and an offset to which address will be jumped. The other class are return instructions reti and reta to return from functions and interrupts.

2.3.4 Secure Intermittent Architecture (SIA)

SIA [5, 26] is a security architecture for IoT devices that harvest their power from the environment, which can lead to power loss if there is a lack of resources to harvest from. When the device runs on solar power and the sun is not shining, it loses power when its battery is empty. These devices practice intermittent computing [28], where they run in a cycle of storing power until they reach a threshold, start execution, run out of power, and need to wait until recharged again. SIA aims to create some periodical snapshots of the runtime state of the device, which would allow the device to start the execution after a power loss from the last snapshot. In order to reach their goal, they provide a security architecture that implements isolation, self-attestation, remote attestation, and secure communication. On top of this, they aim for low-cost security features already present in off-the-shelf microcontrollers like an MSP430 processor, which they used for their implementation.

The main goal of this architecture is to support secure intermittent computing. However, SIA aims to run enclaves in a secure environment and connect the device to the network so they can get software updates. As SIA is implemented on an MSP430 MCU, they use the security features of this device. They isolate enclaves using IPE. Furthermore, they provide attestation to check the state of the updated software. This can be seen as TEE.

2.4 Side-Channel Attacks on Sancus

A microcontroller has a specific architecture, hardware design, and instruction set. All these components have specific effects when they are executed. It takes a specific amount of time to execute an instruction, or a peripheral consumes a specific amount of power in a specific mode. When these characteristics give an indication about the state of the MCU, they can leak information [51]. We will discuss two side-channel attacks that are applicable on Sancus.

2.4.1 Nemesis

Nemesis [49] is a side-channel attack based on interrupts. The goal of the adversary is to determine the control flow of a program based on whether a conditional jump is taken or not. Even when memory is well protected and no data can be leaked, an interrupt can leak some information about the state of an enclave.

A CPU gets halted after each instruction that finished its execution to check if there are any pending interrupt requests. This means the time an interrupt request has to wait before it gets handled depends on the time, the amount, and the current instruction needs to finish its execution. Because different instructions have different amounts of clock cycles, the time between the interrupt request made and handling the first instruction of the ISR will be different depending on the currently executing instruction. If the interrupt is timed at the start of the execution of an instruction, the time the instruction needed to execute is known.

This allows an adversary to determine whether a conditional jump is taken or not by looking at the time the instruction after the potential jump needs. This only applies if instructions have a different execution length.

2.4.2 DMA-Attack

The DMA attack [1] is based on direct memory access. DMA support is used in a system to make it more performant. Instead of a processor waiting hundreds of clock cycles on a response from memory, the processor sends a memory access request to the memory and continues executing. The memory will send an interrupt to the processor when it finishes the request. DMA capabilities alone do not suffice for a DMA attack. There must be contention between the CPU and peripherals. The openMSP430 system offers DMA support. By lacking a DMA controller, only one peripheral device can send access requests to the memory. Sancus prioritizes memory access requests from the CPU to prevent DOS attacks originating from a peripheral. When the CPU and a peripheral send a memory access request at the same time, the request from the peripheral will be delayed, and the peripheral is aware of this delay.

Every instruction has its own pattern of accessing the memory. If the peripheral can intercept all the requests from the CPU, it is possible to know if a conditional jump is taken based on the instructions and pattern of accessing the memory. If the attacker has access to the source code, the attack is more straightforward. Only one execution would be necessary. Furthermore, the DMA request from the peripheral

can be limited to the clock cycles where the CPU would access the memory in one of its branches. Otherwise, the attacker would need multiple executions to figure out at which time points the peripheral needs to issue access requests.

In Sancus reads from and writes to the main memory happening in one clock cycle. If the CPU and peripheral send a request at the same time, the request from the peripheral is delayed by one cycle. The delay of 1 cycle means that the memory is accessed once by the CPU. Assume that the CPU and the peripheral send a request at the same clock cycle, and the CPU makes three more requests in the next three clock cycles. This result in a delay for the peripheral of 4 cycles. In this case, the peripheral knows that the CPU accessed the memory four times. There is no way to say if the CPU read from or wrote to the memory.

The main memory exists of 3 partitions: program memory, data memory, and peripheral space. An attacker can access the three partitions, meaning he can see which part of the memory is accessed by the CPU. It is possible to make a distinction between two instructions based on which partition they access. An attacker cannot distinguish all possible control flows. Two control flows with the same pattern of accessing the memory partitions seem identical even if they have different instructions.

In Sancus, an attack would work as follows. The attacker sets the address register to the address he wants to access. By setting the countdown to a chosen value, the attacker can decide when a memory access request will be sent. When the countdown is set on a specific value, every clock cycle countdown will be decreased by 1. The moment countdown reaches zero, an access request is sent. The default value for the countdown is 0xFFFF, meaning the countdown is turned off. The trace register is 16-bit large. It contains the memory access pattern of the CPU for a given memory partition for the first 16 clock cycles. A 1 corresponds to memory access, and a 0 means no memory access.

2.5 Introduction to ROP-Attacks

One of the most known ROP vulnerabilities is buffer overflow attacks [40]. A buffer overflow occurs when the data loaded into the stack is larger than the space that was allocated for that data. This has as a result that the data overwrites previously stored data on the stack. The stack contains runtime data like variables and return addresses for functions. An attacker can use a buffer overflow and try to overwrite return addresses with as goal of executing a snippet of code. This is a rather old technique as the first buffer overflow attack was the Morris worm[25] from 1988.

The execution of random snippets of code is what defines an ROP-attack [4]. There are two important steps for this attack. The first part is identifying these snippets of code, called gadgets. It is important to know what the gadget does when it is executed and at which memory address the gadget is stored. In more complex attacks, one gadget might not be sufficient, and multiple gadgets must be executed in a sequence. The attacker needs to ensemble an ROP-chain where gadget after gadget gets executed. The second step is finding a manner how to execute these gadgets. The buffer overflow is one way of executing a snippet of code, return-into-libc [47] is

another example where the attacker tries to modify the return address so that he can execute a gadget from libc, the standard C library, which is loaded in almost every program.

Chapter 3

Comparative Analysis

Sancus is a solid choice as secure architecture to provide a TEE regarding the security guarantees it provides with Nemesis [49] and DMA-attack [1] the only two side-channel attacks available. A disadvantage is that a new chip must be designed when a company wants to use Sancus. For some companies, it would be more interesting if some off-the-shelf hardware could be leveraged to level the security guarantees provided by Sancus. As Sancus is built on top of OpenMSP430 [7], analyzing the MSP430 series from TI and their onboard security features can give further insight into the security guarantees.

The specific MCU used for this analysis is the MSP430FR5969 [15, 14, 16]. This Technology has a built-in IPE section to protect a certain part of the memory that, at first sight, has some similarities with a Sancus software module.

This chapter presents a comparative analysis between MSP430 IPE and Sancus software modules. Section 3.1 summarizes the security objectives of TI MSP430 IPE. Section 3.2 provides an attacker model for MSP430. Sections 3.3 and 3.4 discuss respectively isolation and attestation in MSP430 compared with Sancus. The DMA capabilities of MSP430 and Sancus are compared in section 3.5.

3.1 Security Objectives

Security measurements are in place to protect parts of the system or to repel attacks. With IPE, TI creates a protected segment in the FRAM. The security goals are not clear, several claims are scattered around different papers [37, 14, 24, 18]. We will discuss some quotes from TI:

> "IPE module protects a programmed portion of memory from read or write access from anywhere outside of the IP Encapsulated area, even by JTAG [24]."

> "Execution of this portion of memory can be limited to specific callback functions that are defined at the time the IPE module is enabled [24]."

> "When enabled, the IPE module can be used to protect critical pieces of code, configuration data, or secret keys in FRAM memory from being easily accessed or viewed [18]."

These three quotes refer to protection of code and data stored in the memory of a microcontroller. The first quote implies that unprotected memory cannot access the memory segment protected by IPE, which is similar to unprotected memory cannot read or write a software module in Sancus. The second quote limits the execution to the start of a function inside the IPE, preventing an attacker from executing an IPE function from the middle. The third quote points out that it is not only code that can be protected but data as well.

From these quotes, we can conclude that IPE targets to keep both code and data confidential from unauthorized access, make sure the code cannot be modified, and preserve the intended control flow of the functions stored in IPE. However, the following quote assumes there are some issues regarding these security objectives, and there might be some potential leakage of secrets.

> "The IP Encapsulation is only as secure as the code stored in it - poor code security practices can make the code more vulnerable even if it is within the encapsulated area [37]."

> "The IP Encapsulation is only as secure as the code stored in it - poor code security practices can make the code more vulnerable even if it is within the encapsulated area. Code that can read/write to addresses, or that does not follow robust coding practices, is especially concerning [37]."

From the quote above TI seems to be aware that there are some security limitations within IPE. As the code stored in the IP Encapsulation has an influence on the security, this implies that there is a weakness in the IPE. Especially when in the next sentence where they refer to code that can read or write to addresses, which are interesting gadgets for a ROP-attacker.

When it comes to in-field firmware updates, TI is more clear about their security goals [21, 8]. The goal is to protect in the following manners. They want to prevent the firmware image is modified before it can start its execution. Secondly, protect against reverse engineering the firmware image. Any unauthorized image cannot be stored on the device, and an authorized image cannot be stored on an unauthorized device. Overall, the goal is to protect the firmware image during the transport and loading phase.

Security Intentions of IPE When looking at the example code for IPE [37] from TI, You could conclude that IPE is used for protecting confidential code. However, SIA [5] and even the crypto-BSL [8] use the IPE section in a more extensive manner. They both run an entire enclave inside IPE and not only the confidential code.

Number of Enclaves When comparing the overall design of the IPE section, the fact that only one enclave can be protected on an MSP430 board by the IPE contrasts with the multiple software modules that can be deployed with Sancus. Therefore MSP430 has no security goals for IPE sections interacting with each other. This limits the range for the MSP430 to applications where one enclave suffices. However, the interaction between protected and unprotected memory can still be compared.

Shortcomings The recommendations [37] clear register R4 to R15, leaving R0 to R3 uncleared with R0 the program counter, R1 the stack pointer, R2 the status register, and R3 a dummy register. R0 cannot be cleared as the program counter is necessary for proper execution. The stack, on the other hand, can still leak some information. The status register can leak some information based on flags that are set during the execution of IPE when these flags have not been cleared before leaving the IPE section. R3 is a dummy registers and is less important. Applying these recommendations is the programmer's responsibility. It is not a user-friendly way of applying security measurements.

Overall Security Goals Sancus Sancus states their security goals in a clear way [33]. The important ones to compare with are isolation and attestation. Isolation assures that a software module's text and data section cannot be read or written from unprotected memory and the software module can only be executed starting at the entry point. Furthermore, attestation lets a software provider assure that the software module is loaded unmodified. Secure communication between software provider and module and modules between themselves are out of the scope of this thesis.

3.2 Attacker Model

An attacker is assumed to have strong capabilities [38] to break the IPE. The attacker can install machine code on an MSP430 microcontroller and execute it. In particular, an attacker can upload code to the FRAM through the BSL-protected commands. However, he is not directly able to tamper with the IPE section. An attacker cannot tamper with the BSL either, as this is stored in ROM [19].

An attacker can control the network used for communication between the software provider and the device. They can get into a man-in-the-middle position and sniff or modify the network traffic.

Nisarga et al. [32] summarized different threats on MSP microcontrollers and grouped them into security concerns. Our attacker model would focus on non-invasive attacks where we do not break the microcontroller physically open. The focus is rather on-chip level - targeting runtime, memory, and peripheral vulnerabilities - than board-level - power supply, debug interface, and serial interface - vulnerabilities.

From the perspective of the software provider, figure 3.1 shows in green the trusted components and in red the untrusted components. The BSL is a software component

that needs to be trusted to be able to upload the software to the microcontroller. The IPE is seen here as a software module instead of a hardware module as it represents an enclave rather than the hardware component enforcing the security objectives.

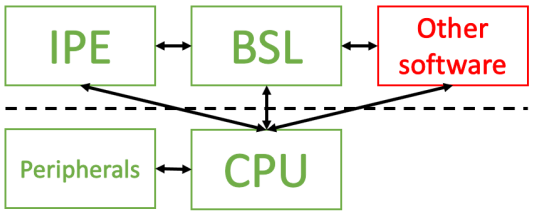

Figure 3.1: Attacker model of MSP430. Green components are trusted, red components untrusted. Components above the dashed line are software components, under the dashed line are hardware components.

3.3 Isolation

MSP430 IPE TI provides some of its MSP430 micro controllers with a memory protection unit. For some models this implements the IPE security feature. TI claims that IPE provides an extra layer of security on top of JTAG in order to improve code protection [37]. Not only should it prevent write or read access, but it should limit the execution of this area to specific callback functions [24].

To reach their security objectives, TI provides a program counter based isolation mechanism and has user recommendations to clear general purpose registers and registers from peripherals to prevent further leakage. These recommendations [37] need to be part of the users programming habits and are not enforced.

A segment of the FRAM can be allocated to a section that is protected by the IPE, the remaining part of the memory will be unprotected. Table 3.1 summarizes how these sections interact with each other, meaning if they can read write or execute other parts of the memory. These results were experimentally verified. As TI claims, there cannot be written or read from outside of the IPE to the inside of the IPE. However an execution can be started from every byte inside the IPE this is contradicting with the limitation to specific callback functions [24].

Hardware Enforcement The manual states that the IPE is divided in different sections [37]. This suggests that functions, constants and interrupts could be protected in a more specific manner regarding their purpose, as they do not need to be written, contrasting from variables. However it is possible to overwrite the value of a constant by writing to the address of the constant in assembly language. This means that the entire IPE segment is handled in the same manner when stored in memory as shown in table 3.1. Assigning different access rules to different section within the same IPE, would allow to optimize the security. The fact that the function section is writeable, allows an attacker to overwrite the executed code. As it is not the intention of a

From/To		Entry	Text	data	Unprotected
Sancus	Entry	r-x	r-x	rw-	rwx
	Text	r-x	r-x	rw-	rwx
	Unprotected/Other SM	- -x	- - -	- - -	rwx
MSP430	IPE	rwx			rwx
	Unprotected	- -x			rwx

TABLE 3.1: Memory access control rules from Sancus 2.0 [33] software module and MSP430 IPE.

programmer to overwrite code in this section, separated access rules for this section could guarantee the integrity of this function section.

The rules from table 3.1 are enforced in hardware on a program counter based manner [14]. This enforcement isolates the IPE from the unprotected memory. Both Sancus and MSP430 implement this isolation with a program counter based hardware solution. As regarding Sancus, a software module is more fine grained by dividing it in a text section and a data section and by providing the text section with an entry point. This contrasts with the section of memory presenting the IPE where no difference is made between the possible types of data that are stored.

This finer granularity creates more specialized protection based upon the content stored in the section. First of all it has an impact on how these sections could interact with each other. As code and constants may not be changed at runtime the text section and entry point cannot be written. In the same way of thinking, as runtime data and variables do not need to be executable, the data section cannot be executed. The program counter based access control is limited to the text section and entry point because under proper conditions the program counter would never be located inside the data section. This is in contrast with IPE with one section that covers the entire memory range of the IPE. As code should be executable and data should be readable and writable, the entire section must allow complete access to the entire section.

A second difference is the interaction with unprotected memory. In Sancus, a finer division can give more specific access rules. As the data section is only meant to be accessed by its corresponding text section, unprotected memory or other software modules cannot access the data section. To enforce that a text section is executed from the start only the entry point is executable. This way the unprotected memory cannot jump in from the middle of the text section, which prevents ROP-attacks. The same reasoning can be applied for IPE as above, due to having one large section and code must be executable from unprotected memory, the entire section must be executable. This means that unprotected memory can jump to the middle of a function or even execute content stored in the variables, constants or interrupt handlers.

The Stack The data section of a software module in Sancus stores the stack of that software module. This protects against leakage of popped elements on the stack or unprotected memory modifying the stack. IPE uses the global stack, which is located in the unprotected memory. If not properly cleared, unprotected memory can read the stack and modify or add elements. However, TI seems to be aware of the problem:

> "More code security measures may be needed for different applications; for example, clearing RAM that was allocated in the course of the function [37]."

The first part does not mention what kind of security measures they intend or what these security measures should protect against. However, the second part of the sentence could point to the fact that the stack is stored in RAM, and the programmer should be aware that the IPE section could push content to the stack. What they do not mention is how to clear the stack. Clearing the entire RAM would remove the entire stack, which would create problems for other functions relying on the stack.

From experimentally findings, not stated by TI but they use this practice in the example code as well, is that IPE does not use the stack to store variables. All necessary variables are initiated as a variable in the IPE section, with as result that every variable is allocated to a memory address inside the IPE section and the IPE will not put variable data on the stack. This must be a practice of the programmer and is not enforced in any way. It is possible to store variables on the stack. Return addresses from function calls or interrupt requests are still stored at the stack.

Other countermeasures can be taken as well, as the usage of the stack is a software consideration. A private stack could me added inside the IPE section. This stack can only be used by the IPE section and is therefore as isolate as the IPE code itself. This would be a similar solution as Sancus where the stack of a software module is stored inside the data section.

Interrupt Protected Code Both Sancus and MSP430 can enable or disable interrupts. However, it is hard to disable interrupts if the code is interrupt-driven. In this case, it is crucial to minimize information leakage as much as possible. Sancus uses a stub that is being executed before jumping to the interrupt handler. This stub clears all general-purpose registers, with as result, the interrupt handler cannot read the registers. After the interrupt is handled, the state of the register from before the interrupt is restored, meaning the interrupt handler could not have affected the registers used by the code from the software module. Such a mechanism is not present on an MSP430 MCU. An interrupt handler can read the data stored in those registers and modify their content before continuing the code of the IPE section.

Interrupt Vector Table (IVT) The interrupt vector table is a memory partition that assigns interrupt handlers to interrupt vectors. An interrupt request triggers an interrupt vector. As a result, the interrupt handler to which the interrupt vector points will be executed. The interrupt vector table is overwritable 3.2, meaning the

addresses where the interrupt vectors are pointing to can be changed at runtime. An interrupt vector pointing to an address from an interrupt handler located inside the IPE can be changed to an interrupt handler located outside the IPE.

> "Any interrupt service routines (ISRs) used by the IP encapsulated code should also be placed in the IPE area so that they are also encapsulated [37]."

From this quote, TI recommends to store all interrupt handlers used by the code inside IPE inside the IPE as well to protect them the same way as the code. This allows an attacker to create a custom interrupt handler and overwrite the interrupt vector in the IVT in such a way that it points to the start of the custom interrupt handler. With as consequence that when the code from within the IPE request an interrupt, the interrupt of the attacker will be executed and no longer the interrupt from the IPE section itself.

The interrupt vector table is located in memory from address 0xFFFF to address 0xFF80 [14]. The IPE can be moved over the IVT in an attempt to protect the interrupt vectors that point to interrupt handlers that are located inside the IPE. However, the IVT has global read and write access, meaning that even if it is inside the IPE, it is still possible to read and write the IVT, which was experimentally verified as well.

One could argue that an attacker can insert a malicious snippet of code inside the IVT to extract a secret of the IPE. When the IVT is located inside the IPE, it has access to the entire IPE section. If an attacker inserts the following:

$$\textbf{mov.w} \quad \&key, \quad R15$$

where $\&key$ refers to the address where the key is stored, the instruction would move the value stored at the location of the key to R15. However, the IVT cannot be executed [14], as shown in table 3.2.

	From/To	Entry	Text	data	IVT	Unprotected
Sancus	Entry	r-x	r-x	rw-	rw-	rwx
	Text	r-x	r-x	rw-	rw-	rwx
	Unprotected/Other SM	- -x	- - -	- - -	rw-	rwx
MSP430	IPE	rwx			rw-	rwx
	Unprotected	- -x			rw-	rwx

Table 3.2: Memory access control rules from Sancus 2.0 software module and MSP430 IPE interacting with the IVT.

Use Case SIA SIA, as discussed in section 2.3.4, uses the IPE section to mimic a software module [5] based on the isolation properties IPE provides. In addition, they disable interrupts, clear the registers of the CPU as well as the registers from the peripherals, and they clear RAM, meaning they follow all recommendations from TI [37]. However, the vulnerability that the whole IPE section is executable cannot be prevented with these measurements. The fact they store all the keys and cryptographic engines in the IPE also gives an adversary even more gadgets to execute.

MSP432 TI provides another series of microcontrollers, namely MSP432, where they implemented IP protected zones [9]. Table 3.3 shows the access rules of an IP protected zone.

An IPE protected zone starts with a section - in table 3.3 this is called the entry section - that sets the Secure Zone Data Enable bit. This allows two defined states for the IP protected zone. If this bit is not set, the entry and IP protected zone cannot read or write to each other and themselves. When the bit is set, these access rights are granted. Both of these states are represented in table 3.3. To transition between previously mentioned states, it is necessary to execute the unlock sequence - the entry - and the re-lock sequence at the end of the IP protected zone. However, it is still possible to start executing the IP-protected zone from the middle. If this unlocking section is not executed, the executed code from within the IP protected zone will not be able to read or write to its own zone. When the execution sequence comes to the end of the IP-protected zone, it re-locks the zone by clearing the Secure Zone Enable bit with as a result, the state of the IP-protected zone changes from the second state to the first state. Access from unprotected memory or other IP-protected zones has the same access rules as an IPE section from MSP430.

Compared to MSP430, from the moment that the Secure Zone Data Enable bit is set, there are no differences between the two. However, the necessity to set this bit to enable reads and writes within the section is an extra difficulty that a ROP-attacker must add to his ROP-chain. A ROP-chain consists of multiple gadgets found in the victim's code. The ROP-attacker creates an executable sequence from these gadgets, called the ROP-chain.

MSP432 comes already closer to Sancus because of the entry point section - meaning a finer granularity. The main difference is that unprotected memory can still execute the entire IP protected zone. As this could be limited in hardware to only the entry section - as essentially done in Sancus - the read and write access in the IP protected zone itself would be less of a problem. Nevertheless, the programmer must still be aware that from within the IP protected zone, data can be executed, and code and constants can be overwritten.

3.4 Attestation

A software provider can update the software on MSP430 by setting up a connection over UART with the BSL [19]. If the software provider provides the correct password

	From/To	Entry	Text	data	Unprotected
Sancus	Entry	r-x	r-x	rw-	rwx
	Text	r-x	r-x	rw-	rwx
	Unprotected/Other SM	- -x	- - -	- - -	rwx
MSP430	IPE	rwx			rwx
	Unprotected	- -x			rwx
MSP432 (1)	entry	- -x	- -x		rwx
	IP protected zone	- -x	- -x		rwx
	Unprotected/Other IP protected zones	- -x	- -x		rwx
MSP432 (2)	entry	rwx	rwx		rwx
	IP protected zone	rwx	rwx		rwx
	Unprotected/Other IP protected zones	- -x	- -x		rwx

TABLE 3.3: Memory access control rules from Sancus 2.0 software module, MSP430 IPE and MSP432 IP protected zones.
(1) configuration bit is not set.
(2) configuration bit is set.

of the BSL, he can access the BSL commands. The software provider can store new code in RAM or FRAM using the TX data block command. After storing this code, the software provider can attest the code by requesting a CRC checksum of the entire program application code. The CRC checksum can be requested with a protected command of the BSL. When this checksum is equal to the generated checksum of the software provider, the code is unmodified. In Sancus, a software provider uploads a software module. After the software module is stored in the memory, the software provider can send a nonce that the software module will use for an encrypt instruction which results in a key. The software module sends the MAC, a part of the key, back to the software provider to check if the software module is unmodified. These are similar methods

When it comes to updating the IPE section, the BSL must perform a special mass erase on the entire memory to clear the IPE settings [46, 18]. From this moment a software provider can load a new IPE section on the MCU. The software provider can attest the state of the IPE section before closing it, as otherwise the BSL cannot longer access the IPE section. Between the time the check has happened and the the IPE section must be closed, an attacker can still modify the code with one of the BSL commands. This manner was created for software updates without IPE. When doing this with IPE, it is a complex and inefficient method as the entire memory needs to

be cleared. TI stated [10] that the in the IPE only confidential code can be stored that will not be modified during the lifetime of the device, while the non-secure part can get software updates.

Later on TI designed the crypto-bootloader [8]. This customized bootloader allowed to update software without the use of the BSL. As the crypto-bootloader is implemented inside the IPE it is possible to perform software updates inside the IPE as well. This paved the way for custom update and attestations mechanism like SIA [5], where an enclave in IPE exists of the application program and the mechasm for software updates and attestation.

Crypto-Bootloader The crypto-bootloader [8] is a customized bootloader designed for MSP430FR59xx and MSP430FR69xx microcontrollers. This bootloader adds countermeasures by using cryptographic primitives and adding countermeasures at the protocol level. As MSP430 is designed for its low power benefits, only symmetric cryptographic algorithms are used as they consume less power than asymmetric algorithms. To provide confidentiality, they encrypt the payload. In order to provide authentication and integrity, message authentication codes are added. At the protocol level, a key can only be used to encrypt packets of one firmware image. This prevents downgrading attacks like the downgrade attack on Trustzone [3] where old versions or, in the case of the BSL old images, can be reuploaded because the same key is used for several image uploads. Therefore the crypto-bootloader provides a method to update the key, and a new image can be authenticated with that new key. After deploying an image, a new key must be provided for the following image. The crypto bootloader does not allow interrupts to ensure that the new image is entirely stored in memory before being executed. Therefore the incoming packets contain a field that contains the number of packets that the images consist of.

Crypto-bootloader is only a solution for microcontrollers that support IPE capabilities as the code, and the keys of the bootloader are stored inside IPE. As mentioned in section 3.3, MSP432 supports IP protected zones. An implementation of the crypto-bootloader for MSP432 would be very similar to this bootloader. Other microcontrollers of the MSP430 series that do not support an IPE section or a similar concept that can securely store the keys cannot use the crypto-bootloader.

Remote Attestation Sancus The crypto-bootloader offers a similar concept and a similar implementation as remote attestation in Sancus. However, crypto-bootloader focuses its solution on the incoming messages and attests those. It relies on the fact that the crypto-bootloader cannot be interrupted, and therefore the code will be stored unmodified in memory. Sancus attests the in-memory located code to check if it is loaded unmodified in memory. It does not rely on a secure loading procedure like MSP430.

Time-Of-Check-Time-Of-Use Time-Of-Check-Time-Of-Use (TOCTOU) refers to the difference in the concept of time between the fixed time point where you perform a software provider remotely attests the state of the code and the time

period that the code is used. Even when a software provider executes multiple remote attestations, the software does not know the state of the code during the period between two attestations. If an attacker modifies the code between two remote attestations, the modified code can be used until the next attestation.

Nunes et al. provide two possible solutions to protect against this vulnerability [36]. The first solution is a hardware module that checks if writes happen within an enclave. If this is the case, these modifications are logged within a protected memory region in the hardware module that is unwritable for the MCU and DMA. If this modification has a different timestamp than the last authorized modification, the hardware module can verify that it is an unauthorized modification. The second method uses a hardware module that stores the challenge provided by the software provider when they do a remote attestation. This challenge is only stored if the enclave is modified since the last stored challenge and the challenge was successfully authenticated.

Use Case SIA SIA creates an attestation method [5] similar to the attestation of Sancus. In fact, they provide two different remote attestation methods. A slow attestation uses a challenge from the software provider and the entire memory of the module, and a fast attestation only uses the challenge. They both generate an authentication tag based on the key of the module. The software provider can check if the module is unmodified with this tag. An extra feature is self-attestation, where the module recomputes its key and compares it with the actual key of the module. The attestation protocols are secure and reliable because they are located in the IPE section, and the isolation guarantees that the IPE section provides. In section 3.3, we discussed some flaws in the isolation mechanism of the IPE section. It is questionable if these attestation protocols are actually secure.

3.5 DMA Capabilities

OpenMSP430 implements a DMA interface whose primary goal is to handle DMA requests from peripherals so the CPU can continue its execution [7]. The difference between a DMA interface and a DMA controller from MSP430 is subtle as the DMA controller's primary purpose is to handle DMA requests from peripherals in order to keep the CPU in a low power mode [17]. Consequently, the CPU halts its execution when the DMA needs to handle a request to make the bus line accessible. The DMA interface of Sancus can work in parallel with the CPU, meaning the CPU does not get halted. The DMA controller has three channels that can be independently configured, as explained in section2.3.1, but the DMA controller handles only one request at a time. Sancus' memory consists of three partitions where only one request can be configured at a time for each partition. However, the DMA interface can handle DMA requests to different partitions in parallel.

In MSP430, a peripheral cannot directly connect with the DMA controller. Communication must take place over I^2C in order to set the source and destination address. Sancus, on the other hand, has an address and data-in wire to which the

peripheral can connect. This design makes DMA requests in Sancus more dynamic as MSP430 has the intention to set the addresses only once, which is a more static approach. However, the range of addressing modes and transfer modes makes it possible to easily swap the address. An example could be a sensor that wants to store its measured value to the FRAM, the destination address is initially set to a fixed address, but because of the transfer and address mode, the destination address can be increased with every DMA request that the sensor issues. One could say that the DMA interface in Sancus is a general-purpose DMA, while the DMA controller in MSP430 is a specialized DMA.

This has consequences for a possible DMA attack [1] in MSP430. As the source and destination addresses of the DMA request are stored in the proper registers, only a trigger is needed to start the transfer. When a DMA transfer is triggered, the CPU finishes its current instruction, and then the CPU gets halted until the DMA transfer is finished. At this point, the CPU and DMA controller has no contention. In Sancus, it was this contention that made it possible to execute a DMA attack as the DMA interface was delayed when the CPU accessed the memory at the same time. In MSP430, the DMA controller has to wait until the CPU has finished its instruction. The DMA controller cannot detect if the CPU was accessing the memory in the meantime.

Even if there would be contention, it is impossible to check whether the transfer is finished or not, whether Sancus has a signal where a peripheral can check if the request has finished. As the CPU and a peripheral trigger the DMA controller simultaneously, one request would be executed before the other. However, it is impossible to tell which one is executed first as the CPU will only continue its execution when both requests are finished.

However, the halting of the CPU creates opportunities. The DMA controller can be used in the same way as timing an interrupt as Nemesis does, as the trigger can only start the DMA request as the CPU has finished its current instruction. The power of a DMA request is smaller than that of an interrupt in the sense that a DMA request is bound to its transfer and address modes to read or write data.

Table 3.4 provides the access rules of Sancus and MSP430 and, additionally, a DMA controller. Their rules are similar as the DMA cannot access the protected areas and can read and write to the unprotected areas. There might be no contention between the IPE reading or writing within its section and the DMA controller. However, there is contention between the IPE reading or writing outside of its section and the DMA controller as the stack is located outside the IPE section. Another difference between interrupts and DMA is the general-purpose registers. A DMA cannot access those - as these are not part of the memory - as an interrupt does have this access.

3.6 Conclusion

This chapter defined security objectives and an attacker model for TI MSP430 IPE based on the documentation TI provides.

From/To		Entry	Text	data	Unprotected
Sancus	Entry	r-x	r-x	rw-	rwx
	Text	r-x	r-x	rw-	rwx
	Unprotected/Other SM	- -x	- - -	- - -	rwx
	DMA(*)	- - -	- - -	- - -	rw-
MSP430	IPE	rwx			rwx
	Unprotected	- -x			rwx
	DMA	- - -			rw-

TABLE 3.4: Memory access control rules from Sancus 2.0 software module and MSP430 IPE interacting with DMA.
(*) Sancus 2.0 is extended with DMA capabilities.

Furthermore, we compared the isolation and attestation guarantees MSP430 and Sancus give. Differences in the hardware design can have severe consequences. Differences on the software level were pointed out as well.

In the end we compared the DMA capabilities between MSP430 and Sancus, where we concluded that due to the difference made in the hardware design, a side-channel DMA attack is unlikely on MSP430.

Chapter 4

Experimental Results

This thesis uses three key features to exploit the IPE section. The first one is code reuse features, where the attacker can execute snippets of code from within the IPE section. The second feature is interrupt handlers, as interrupts have the capability to pause the control flow of the IPE section to execute some attacker code or tamper with the registers of the CPU and peripherals. The last feature is the DMA controller, which is similar to the previous feature as it can write to memory addresses but not to the general-purpose registers from the CPU and cannot execute code.

These features can be used in various manners to break the control flow of the IPE section by tampering with registers, the stack, and the IPE code itself with as goal of extracting a secret, zeroing out a key, or determining the control flow of the IPE section.

This chapter dedicates a subsection to each of these features and an end-to-end attack on the example code of TI where a combination of these features creates a successful attack.

4.1 Security Weaknesses

In this first section, we discuss opportunities for an attacker to break the security features of MSP430 based on table 4.1. These opportunities can be attack primitives or some valuable tools to scale the range of attack primitives.

From/To		Entry	Text	data	IVT	Unprotected
MSP430	IPE	rwx(1)			rw-	rwx(4)
	Unprotected	- -x(2)			rw-(3)	rwx
	DMA	- - -			rw-	rw-(5)

TABLE 4.1: Memory access control rules from MSP430 IPE with indexes to security weaknesses.

(1) Same access rules for entire IPE We discussed in section 3.3 why it is helpful to separate text from data. However, as this is not the case, an attacker who can execute variables or modify the stored code is a valuable feature to consider when looking for gadgets to execute a code reuse attack, which is further discussed in section 4.2.

(2) Unprotected memory executes IPE As unprotected memory can start an execution from every address inside the IPE. This raises several opportunities for the attacker. This allows code reuse attacks, as explained in section 4.2 or can be used to bypass security measurements, which is featured in section 4.3.

(3) IVT Accessibility The IVT is accessible from unprotected memory, independent of its location, inside or outside the IPE. As an attacker, this is a valuable feature to keep in mind to replace interrupt handlers located inside the IPE with interrupt handlers controlled by the attacker.

(4) IPE stores on the global stack One of the reasons why IPE accesses unprotected memory is that default IPE uses the global stack. The attacker controls this stack and gives extra features to the attacker as he can modify return addresses or values from variables stored on the stack. It is worth mentioning that this behavior is not enforced. A programmer could set up a stack inside his IPE section. Different from other topics in this list, this is a software problem that can be prevented. The other topics in this list are a result of the hardware design.

(5) Halting the CPU As the DMA controller halts the CPU when handling a DMA request, as explained in section 3.5. This provides a different method to interfere with the connection between IPE and unprotected memory.

4.2 Code Reuse Attacks

Code reuse attacks [4] reside from the fact that the entire IPE section is executable. An attacker needs to search for valuable gadgets - an instruction or sequence of instructions - inside the section. This differs from a typical code reuse attack, as there is no return instruction needed to modify the control flow. The attacker can call any instruction from within the IPE section with a call instruction.

Gadgets

A code reuse attacker executes snippets of code to reach his goal. These snippets must be stored inside the IPE section to have the same access rules as the IPE section. When looking at table 3.1, these snippets have full access over the entire IPE domain, and an attacker can execute these snippets from outside the IPE. The attacker should specifically look for instructions that interact with registers as these are not cleared before executing a part of the IPE section as explained in section 3.3. This is a straightforward manner to insert well-chosen values inside the code.

Finding Gadgets

The first place to look for these gadgets is the disassembly created when compiling the code in code composer studio, which uses MSP-CGT [12]. This disassembly - the binary file that represents the values in memory translated to instructions- shows the code and how the microcontrollers store it with the instructions based on how the code would be executed as intended. The appearance of these gadgets is highly dependent on the specific content of the implemented code. However, there are several methods to modify the assembly underneath the code or modify the assembly stored in memory:

Different Compilers Different compilers may result in a slightly different compiled version of the same code. This small modification might be enough to create a gadget that could be exploited by an attacker. Some example compilers are msp430-gcc [22], msp430-elf-gcc [22], and the clang cross compiler [45].

Optimization Levels A compiler can optimize the source code in terms of performance, size, or readability, and for each of these, there are several optimization levels. Depending on the selected optimization, new gadgets can be introduced that could be helpful for an attacker to create an ROP-chain.

Different Start Addresses An attacker is not limited to the code from the same addresses as is suggested by the original disassembly. He can, for example, start the execution from an address that in the actual execution would be interpreted as an argument from an instruction. Therefore, it is interesting to create disassembly files from different offsets from the original binary file. Not only the first instruction will differ from the original disassembly, but an entirely new sequence can be created. This is because the first instruction has a different instruction length compared with the length minus one of the first instruction from the actual disassembly. The next instruction will start at a different address than in the actual assembly, which results in a new instruction. As long as this is the case for every next new instruction, a potentially new gadget can appear. From the moment one instruction starts at an address where a new instruction starts in the original disassembly, all following instructions will be the same as the actual disassembly.

Overwriting Code Overwriting code is a result from table 3.1, as the entire IPE section is writable. An attacker could find gadgets that can be used to modify the original code. These gadgets do not contribute directly to the attack. When using this method, the attack exists of two stages. The first stage is using the ROP capabilities on a gadget that can modify the code - this is what is described in this section. The second stage is using the ROP capabilities on the code that was modified in the first stage.

A straightforward example could be

$$\textbf{bic.w} \quad \#0x10, \quad @R15$$

Addresses	0x4814	0x4816	0x4818
Instruction	**mov.w**	# 0x50	& 0x4832
Memory value	40B2	0032	4832

Table 4.2: Memory addresses and memory values of an instruction

This instruction will clear the fifth bit of the value located at the address stored in R15. If the attacker stores an address that is located inside the IPE module in R15 and afterwards executes the gadget above, the value stored at the address that is stored inside R15 will be modified. This method allows an attacker to modify the original code. Overall there are two possibilities to investigate with this gadget. The first possibility is overwriting an argument of an instruction. Take the instruction

$$\textbf{mov.w} \quad \#0x50, \quad \&0x4832$$

which would store 0x50 at address 0x4832. If an attacker finds this instruction stored in memory as represented in table 4.2, an attacker can store address 0x4816 in R15 and execute the gadget. This would result in a change of 0x50 to 0x40. Another option is storing 0x4818 in R15. This results in modifying the value to 0x4822 instead of 0x4832. The second possibility is modifying instructions. This would have a larger impact on the code as not only the goal of the instruction will change but the length of the instruction as well. If the attacker modifies the value at address 0x4814 with the previously mentioned gadget, the value becomes 0x40A2, which results in instruction

$$\textbf{mov.w} \quad PC, \quad \&0x50$$

The third word, 0x4832, of the original instruction becomes a new one word instruction

$$\textbf{mov.w} \quad @R8+, \quad SR$$

In this case, the following part of the code remains the same as in the original assembly, as the next word is the start of a new instruction, just as it was in the original code. If, at address 0x4818, a different value would have been stored that would result in an instruction longer than one word, the start of the next instruction would become the argument of this instruction. This effect would ripple through the entire code and could generate new gadgets in the instruction that is modified and potentially in all following instructions.

Code Injection

When no valuable gadgets appear in the disassembly of the IPE section, it might be possible for an attacker to inject his code inside the IPE section. A key feature of this attack is the presence of variables inside the IPE, where the attacker can configure the values of the variables in such a way that they form an executable

gadget. Encryption and decryption are typical applications of the IPE as it can store the keys, and a function that loads plaintext inside the IPE can be present in the IPE section. When this plaintext is temporarily stored inside the IPE section, an attacker could take advantage of this situation and execute the inserted plaintext. If this plaintext is well-chosen, it could leak the key. However, this is a result of bad coding practice of the programmer as the plaintext should be immediately stored in the AES accelerator for the cryptographic operation. This is a clear example to show the possibilities of the attacker. A more realistic and more complicated scenario for the attacker would be to configure a series of different variables in such a way that their values form a gadget the attacker can execute to extract the key. This method differs from overwriting code as it only modifies the values of variables - not the code - and exploits the fact that these variables are executable, as shown in table 3.1.

Use Case SIA

The SIA software is not available. However, they followed the recommendations of TI on how to implement functions in IPE. SIA creates checkpoints from the running IPE module. When the power is restored after a power loss, the microcontroller can restart the execution of the module from the saved checkpoint. This checkpoint is stored in non-volatile memory, but it is not stated if that is inside the IPE or not. Assuming it is stored in the IPE - as this could be a safe way to store the checkpoint - an attacker can overwrite the checkpoint due to the fact of the ROP-primitive.

4.2.1 Increase the Credit

Assume a small IPE module that a user can call to increase the stored `credit`. The `credit` can only be modified once every period. Listing 4.1 represents the increase function of the IPE module in assembly. If the `credit` is not modified yet, the credit is increased by one, and `isModified` is set to one. Furthermore, the timer is configured and started. In the end, the registers are cleared, following the recommendations of TI. If the timer reaches the specified value, the interrupt handler set `isModified` to zero, and the user can increase the credit once again.

Attack The attack is represented in listing 4.2. The valuable gadget found in listing 4.1 is located at address 0x440E. If the attacker starts the instruction from this memory address, the credit will be increased by one, configure the timer, clear the registers and return back to the attacker code. By looping this call, an attacker can immediately increase the credit as much as he wants without having to wait until the credit is modifiable again.

4.3 Interrupt Attacks

An attacker can interrupt code from the IPE section. This is a powerful primitive as this gives an attacker the following capabilities:

```
    increase():
004408: tst.b   &isModified          ; if(isModified == 0){
00440c: jne     ($C$L2)
00440e: inc.w   &credit              ; credit += 1
004412: mov.b   #1,&isModified       ; isModified = 1;
004416: mov.w   #0x0010,&TA0CCTL0    ; enable interrupts
00441c: mov.w   #0xc350,&TA0CCR0     ; set wait time
004422: mov.w   #0x0214,&TA0CTL      ; start timer}
    $C$L2:                           ; clear registers
004428: clr.w   R4
00442a: clr.w   R5
00442c: clr.w   R6
00442e: clr.w   R7
004430: clr.w   R8
004432: clr.w   R9
004434: clr.w   R10
004436: clr.w   R11
004438: clr.w   R12
00443a: clr.w   R13
00443c: clr.w   R14
00443e: clr.w   R15
004440: reta
```

Listing 4.1: Increase.

```
1 void main(void) {
2     int i = 0;
3     for(i = 0; i < 1000; i++){
4         // start execution from address 0x440E
5         __asm(" CALLA #0x440E");
6     }
7 }
```

Listing 4.2: Attacker code code reuse.

Register Access As discussed in section 3.3, if an interrupt occurs, the interrupt handler gets executed, and the registers contain values from the IPE section as these registers are not cleared. This allows an attacker to steal the information stored in these registers or modify the values in these registers. When the code from the IPE section recontinues executing, its behavior is different from what was initially intended.

Change Return Address When an interrupt occurs, the address of the last executed instruction is stored on the stack. The execution of an interrupt handler ends with a return instruction which pops an item from the stack and jumps back to the address that was popped to continue the execution from that address. An interrupt can exploit this to jump to another address than the address where the execution got interrupted. This allows an attacker to partially execute functions from the IPE and jump to another function or gadget or back to the code controlled by the attacker. This behavior can be enforced in two manners. The first manner is modifying the stack pointer to another address on the stack. The second manner is changing the last stored value on the stack to an address of choice.

Determining Control Flow An attacker can perform an attack like Nemesis [49] to determine the control flow of the IPE section. However, the capabilities discussed above utilize more of the resources available and result in more powerful attacks. Those should be considered first.

AES Accelerator A specialized module stands in for cryptographic operations. This module is called the AES Accelerator and can encrypt and decrypt 128, 192, and 256-bit challenges [14]. The accelerator has a register for the key, a register for the input data, and a register for the output. The registers that are used for input will always read out as zero. This means that an attacker cannot target these registers to steal the key.

TI recommends some countermeasures [37] in order to protect the IPE section from these interrupts.

Store Interrupts Inside IPE TI recommends [37] creating interrupt handlers for important interrupt vectors - interrupt vectors of peripherals used in the IPE section - in order that an attacker cannot create an interrupt handler for these interrupt vectors. However, as section 3.3 discusses, the IVT is always writable. As a result, an attacker can define a function and store the start of this function inside the IVT at an interrupt vector pointing to an interrupt handler within the IPE section. When an interrupt occurs for that interrupt vector, the interrupt vector of the attacker will be executed instead of the interrupt vector inside the IPE.

Disable Interrupts A programmer can disable interrupts at the start of an IPE function and enable interrupts again at the end of the function. This only protects

```
1     cmp.w 0x007b, R12          ; if(R12 == 123){
2     jeq ($C$L1)                ; storedValue = R13;
3     mov.w R13, &storedValue    ; }
4 $C$L1:                         ; else{
5     nop                        ; //do nothing}
```

Listing 4.3: Conditional branch.

against interrupts if the function is executed from the start. As section 4.2 discusses, the IPE section can be executed from every location. This means these disabled interrupt calls can be bypassed. When a function has arguments and interrupts are disabled at the start, it is essential to load the arguments in the correct registers before starting the execution after the disabled interrupts call. Another special case could be to minimize the area of code that needs to be protected from interrupts. The disable interrupt call can be bypassed by executing the function and interrupting right before the disable interrupt call would be executed. If the interrupt handler changes the return address to the address of the instruction immediately after the disable interrupts call, the function will continue without the interrupts disabled. This means that disabling interrupts cannot prevent interrupts from happening.

As these countermeasures are not sufficient to protect against malicious interrupts, modifications should be made at the hardware level. If only the start of a function could be executed instead of jumping into the middle of a function, disabling interrupts will work as these calls can no longer be bypassed. A design of the IPE section with an entry point that is executable from outside and a text and data section that is executable from inside the IPE section similar to the design of a software module in Sancus can solve this problem.

4.3.1 Nemesis Attack

As example we will use an IPE segment that stores a variable value protected by a password as shown in listing 4.3. The function-section contains a function that stores the given value in `storedValue` variable which is located in the vars-section, if the provided password equals `storedPssw` which is located in the const-section. The adversary can set up a timer followed by a function call to change the stored value. With the right timing, an interrupt occurs at the first cycle after the `jeq` instruction. If the jump is taken, the next instruction is a `nop` instruction, which takes one cycle to execute. When the jump is not taken, the next instruction is a `mov` instruction that takes four cycles to execute. As explained in section 2.4.1, the adversary can now determine the program's control flow.

4.3.2 Register Swap

Assume a small enclave that contains a variable `credit` that stores a value. A user can store an amount in credit by calling `storeCredit(int amount)`. Listing

```
1  storeCredit():
2      cmp.w    #1,R12          ; if(R12 > 0){
3      jl       ($C$L1)
4      add.w    R12,&credit     ; credit += R12}
5  $C$L1:                      ; clear registers
6      clr.w    R4
7      clr.w    R5
8      clr.w    R6
9      clr.w    R7
10     clr.w    R8
11     clr.w    R9
12     clr.w    R10
13     clr.w    R11
14     clr.w    R12
15     clr.w    R13
16     clr.w    R14
17     clr.w    R15
18     reta
```

Listing 4.4: Store credit

4.4 represent the assembly version of this function. This assembly function has no amount parameter. The parameter is stored in R12. The store credit function checks if the provided amount is larger than zero. If this is true, the amount is added to the credit. Otherwise, nothing happens. In the end, the registers are cleared following the recommendations of TI.

Interrupt An attacker can interrupt this function before line four is executed. The interrupt handler has access to the registers and can modify the value in R12. Afterwards, the function will continue its execution with the modified value in R12. As a result, instead of the value that was originally stored in R12, the value of the attacker will be used for the remaining part of the execution. If the attacker stored -1000 in R12, the `credit` would be decreased by a 1000.

4.3.3 Interrupt Loop

For this attack, we use the same enclave as in listing 4.1. The idea of this attack is to loop over the gadget that increases credit while bypassing the check on `isModified`. The attacker configures the timer and calls the gadget. The timer is configured in such a way that it requests an interrupt while the gadget is executing. This way, the interrupt handler will start executing right after the gadget.

Change Return Address Listing 4.5 represents the assembly code of the interrupt handler of this attack. The interrupt handlers increases the variable `iterator`.

```
1  custom_interrupt():
2      cmp.b    #0x000a,&iterator    ; if(iterator < 10){
3      jlo      ($C$L3)
4      jmp      ($C$L4)
5  $C$L3:
6      inc.b    &iterator            ; iterator += 1
7      pop.w    R15                  ; pop value from stack
8      pop.w    R15                  ; pop value from stack
9      push     \#0x0000             ; push 0x0000 to stack
10     push     \#0x480E             ; push 0x480E to stack
11     nop
12     eint                          ; enable interrupt
13     nop
14 mov.w   #0x0005,&TA0CCR0          ; set wait time
15     mov.w    #0x0010,&TA0CCTL0    ; enable interrupts
16     mov.w    #0x0222,&TA0CTL      ;start timer}
17 $C$L4:
18     reta
```

Listing 4.5: Custom interrupt.

This variable makes sure that the interrupt handlers will be repeated a maximum amount of times, in this case, ten times. The goal of the interrupt handler is to change its own return address back to the address of the gadget instead of the address of the instruction after the gadget. This is accomplished by popping two values on the stack and replacing them with the address of the gadget. When the interrupt handler reaches the return instruction, the execution will continue from the inserted address. As long as the `iterator` has not reached its maximum value, this process will be repeated. Otherwise, the interrupt handler returns back to the address after the gadget and finishes the execution of the remaining part of the function.

4.4 DMA Attacks

Bognár wrote about a novel DMA attack on Sancus [1]. The main reason this attack can succeed is because of the contention between the DMA interface and the CPU. They can try to access the memory at the exact same moment, resulting in a delay of one of the two. In Sancus, the DMA interface will be delayed. It is this delay that lets the attacker determine the control flow of the software module. On MSP430, this contention gets lost because of DMA requests causing the CPU to halt. As the CPU writes to the memory in, for example, a move instruction, the DMA controller can not verify this as the DMA request only gets handled as the CPU has finished the instruction as explained in section 3.5.

The DMA controller can serve another purpose which is similar to interrupts, as explained in section 4.3. However, it is more limited in access range and execution

capabilities. The DMA controller can only modify the stack and registers from peripherals as these belong to the address space. The registers from the CPU are not accessible as these do not belong to the memory address space. The IPE section is not accessible as well, which is shown in table 3.1.

4.4.1 Modify Return Address

This attack is similar to the attack explained in section 4.3.3. Instead of using interrupts to modify the return address, this time, the DMA controller is used to modify the stack. As the DMA controller only halts the CPU and afterwards continues from where the execution was halted, the modified return address is the address where the `increase()`-function returns to. The attacker can use this method to iterate over a part of the `increase()`-function.

Payload The payload the attacker injects into the stack is the source variable in listing 4.6. The content of this payload exists of memory addresses. Each address exists of two words, where the first word in the variable is the second part of the address - meaning the 0x440E and 0x0000 form address 0x0000440E. Address 0x440E refers to the gadgets we want to execute as shown in Listing 4.1. Address 0x10082 refers to the instruction that follows after the call instruction made in listing 4.6 at line 40. This address is meant to return the control flow to the attacker.

Configure DMA Controller In order to inject this payload, the attacker uses the DMA controller. As the payload is several words long, the DMA controller will perform a block transfer where the source and destination will be increased by one every time a word is transferred. The source address is the start address of the payload. The return address is the address on the stack where the `increase()`-function stored its return address. Furthermore, the size of the transfer block is provided, and DMA is enabled. This configuration is represented from line 6 to line 21.

Clear the Stack The attacker clears a part of the stack for a proper execution and puts the stack pointer inside the clear section.

Execution of the Attack It is crucial for the attack that the DMA controller overwrites the stack after the execution of the `increase()`-function has started to make sure that the return address will be overwritten. Therefore, a timer is configured from line 30 to line 33 to trigger the DMA request while the enclave is executing. From the moment the payload is stored on the stack, the `increase()`-function will return to the address of the gadget and execute it to increase the credit until it reaches the last address of the payload which returns the control flow back to the attacker.

```
1  // payload
2  #pragma PERSISTENT(source)
3  const uint16_t source[4] = {0x440E, 0x0000, 0x440E, 0x0000,
4                              0x440E, 0x0000, 0x440E, 0X0001};
5
6  void main(void) {
7      // set DMA trigger to timerA
8      DMACTL0 = 0b0000000000000001;
9      // set transfer mode to block transfer
10     // with increasing source and destination addresses
11     DMA0CTL = 0b0001111100000000;
12
13     // Configure DMA channel 0
14     // set source address
15     __data20_write_long
16         ((uintptr_t) &DMA0SA,(uintptr_t) &source);
17     // set destination address 0x23E8 = address on the stack
18     __data20_write_long
19         ((uintptr_t) &DMA0DA,(uintptr_t) 0x23E8);
20
21     DMA0SZ = 8;            // size of the the block
22     DMA0CTL |= DMAEN;     // enable dma
23
24
25     int i = 0;
26     // clear the stack
27     for(i = 0; i < 15; i++){
28         __asm(" PUSH.W #0x0000");
29     }
30
31     TA0CCR0 = 0x0C;           // set timer for flag
32     TA0CCTL0 = 0;             // set interrupt flag for DMA
33     TA0R = 0;                 // set counter to zero
34     TA0CTL = 0x0222;          // start timer
35
36     __asm(" MOV.W #0x23EC, SP");     // set stackpointer
37     increase();                      // call enclave
38 }
```

Listing 4.6: Attacker code for modifying the stack with DMA.

```
1 mov.b  &IPE_i, R15              ;AESAKEY = key[IPE_i];
2 rlam.w #1, R15
3 mov.w 0x0486e(R15), &AESAKEY
```

Listing 4.7: Key transfer.

Countermeasures The capabilities might be limited, but they are still a potential threat. TI does not state anything about vulnerabilities that could be exploited with a DMA controller. As far as we know, they do not state any countermeasures for these vulnerabilities. In line with the disable interrupt call, there is a DMA enable signal. DMA requests will not be executed when the signal is set to zero. A possible solution could be to disable DMA requests at the start of an IPE function. The same reasoning as in section 4.3 could be made. These DMA disables could be bypassed because of the access rules from the IPE section.

4.5 Zero out the Key

When it is not possible to extract the key from the IPE, another solution could be to modify the address which the CPU fetches the key from to provide it to the AES accelerator. As the attacker knows the key, this allows him to decrypt the ciphertext back to the original message. [6]

Key Transfer The key is stored in an array structure that contains words inside the IPE. In order to transfer the key to the AES accelerator, one loops over the elements in the array and transfers every element to the AES accelerator with

$$AESAKEY = key[IPE\_i]$$

where IPE_i refers to the index. Listing 4.7 presents the assembly equivalent of this operation. Remark that the iterator is loaded into R15 and shifted one bit to the left with the rlam instruction, which is equivalent to multiplying by two. Because the array stores words, this means that an index must be multiplied by two to point to the correct location as in memory is counted in bytes instead of words. The actual transfer happens in line 3. 0x0486e is the starting address of the array that stores the key. R15 holds the offset that is added to the address. The value located add 0x0486E plus the value in R15 is moved to the register of the AES accelerator.

Modify Input of AES Accelerator It is the instruction from line 3 that is a valuable gadget for an attacker as he can inject a value into R15 to choose which value will be used as the key. If the attacker can locate a 0x0000 word in memory and can determine the value for R15 so that the base address of the key plus the value in R15 is equal to the address where 0x0000 is located, he can insert a key that is equal to zero in the AES accelerator.

4.6 End-to-End Attack

TI provides an example project to implement an IPE section [37]. In this end-to-end attack, we try to find vulnerabilities in the IPE section of this project. The IPE section stores a key and a method that transfers this key to the AES accelerator. The goal is to extract this key out of the IPE section. For this attack, the code reuse attack primitive, interrupt attack primitive, and the global stack are combined in one attack.

ROP-primitive As the gadget is stored in the middle of the IPE section, the attacker must be capable of executing snippets of code inside the IPE from outside the IPE as explained in section 4.2.

Interrupt-primitive The gadget is stored in a function that follows the recommendations of TI to clear general-purpose registers of the CPU. If the attacker does not interrupt before the registers are cleared, the content stored in R14 will be lost, and the secret cannot be extracted. To solve this problem, an interrupt can be requested after executing the gadget so the content of R14 can be extracted before R14 is cleared. The IPE section contains an interrupt handler used for the same interrupt as the one the attacker is planning to use. Therefore it is essential that the IVT is overwritable, as explained in section 4.3, so the interrupt vector can point to a self-defined function.

Stack As the code gets interrupted after the gadget, the attacker can immediately go back to the execution of the attacker code instead of the IPE section. With the knowledge that the return addresses are stored on the stack - which is not protected - the attacker is capable of modifying the stack pointer.

Find Gadgets The first step is looking for valuable gadgets. As the binary file from the compiler of code composer studio does not provide interesting gadgets, other compilers and different optimization levels were used in search of potential gadgets. Compiling the example file with the msp430-gcc compiler [22] and optimization level O2 - which is performance level 2 optimization - an interesting gadget appeared:

$$\textbf{mov.w} \quad @R15+, \quad R14$$

Further in this section we will refer to this instruction as the gadget. This gadget moves the value located at the address stored in R15 to R14 and increments the value stored in R15 with two bytes. The concept of this attack is to execute this gadget from outside and provide it with the correct argument for R14 that after the execution, R14 will hold a secret. In order to successfully create this attack, a combination of the following attack primitives and vulnerabilities is necessary.

Interrupt The next step is writing the interrupt handler. In order to do this, two key questions need to be asked. First, we need to know at what point in the control flow the interrupt will occur, as this is important to understand the context of the

```
1 mov.w R15, &buffer   ;move content of R15 to buffer variable
2 mov.w #0x23F8, SP    ;set the stack pointer on address 0x23F8
3 reta                 ;return
```

Listing 4.8: Interrupt handler of the end-to-end attack.

registers of the CPU. The second question is about what the interrupt handler needs to do. The answer to the first question is that the interrupt will occur right after the gadget is executed. This means that R14 will hold the secret that needs to be extracted. For the second question, the target of the interrupt handler is to extract the secret out of the register and let the interrupt handler return to the attacker code instead of the IPE code. The interrupt handler is given in listing 4.8. The first instruction moves the content of R14 to the address of the `buffer` variable, a variable on FRAM that the attacker controls. The second instruction sets the stack pointer back to the location where the address of the attacker code is stored, as the attacker made a call to the gadget. The return instruction will set the program counter to the value that is stored on the stack, which will, due to the change of the stack pointer, be the code of the attacker.

Attacker Code With this building block, the attacker can create the main block of the attack. The attacker needs to first overwrite the interrupt vector location of timerA in the IVT as this is the interrupt vector that the IPE section uses and the vector necessary for the attacker. The result is that the interrupt vector points to the start of the attacker-created interrupt handler. The next is preparing for the call to the gadget. This means that the timer needs to be set to request an interrupt as the gadget is executing, and register R15 should be set as to the location of the key. Now the attacker calls the gadget. As the interrupt handler is built in such a way that it returns the control flow back to the attacker code, the attacker can store the part of the key in `buffer` to an array that can contain the entire key.
Suppose the attacker iterates over this method and changes every iteration the address of the key and stores the content of the `buffer` to the following location of the array. In that case, the attacker can extract the entire key out of the IPE section.

4.7 Conclusion

This chapter provides an overview of several security limitations of TI MSP430 IPE and shows how they could be exploited. Three attack primitives are explained. The first attack primitive is the code reuse attack, where an attacker is able to start execution from every location inside the IPE. The second attack primitive is the interrupt attack, as an attacker can interrupt the execution of the IPE and steal or modify information. The last attack primitive is the DMA attack, where an attacker can halt the CPU and modify information with the DMA controller.

In the end we combined these security limitations to extract the key out of the example code of TI and showed how the security limitations allow an attacker to zero out the key.

Chapter 5

Conclusion

Interconnected embedded devices have seen an immense growth over the last few years. Trusted execution environments were designed to allow these embedded devices to run software in a secure environment that allowed them to run these programs in an isolated manner. However, TEEs come with hardware security modules that need to be mounted on the chip. As embedded devices aim for cost-beneficial practices, off-the-shelf microcontrollers with built-in security features might provide the same security guarantees as a TEE. In this thesis, we investigated if MSP430 has security features that could serve to provide a TEE. To compare the security feature of MSP430, we used Sancus as a standard to measure if MSP430 could provide a TEE.

5.1 Limitations and Future Work

Other Security Features In this thesis, we focused mainly on MSP430 IPE. We have touched MSP432 IP protected zones. A more in-depth discussion can be made with MSP432, similar to the discussion and attack primitives we made for MSP430. TI microcontrollers with security features that give similar security guarantees can be added to the comparison.

Evaluation of Use Cases As MSP430 are commercial MCUs, projects like SIA can be challenged for an in-depth evaluation of how they use the IPE section and their vulnerability to the attack primitives mentioned in this thesis. As SIA's code is not available, other use cases can be found for this evaluation.

Countermeasures Several countermeasures can be added as explained in previous chapters. These can be implemented to provide better security on the software level for IPE, like storing a private stack inside the IPE instead of using the global stack or introducing a stub that clears the general-purpose registers before executing interrupt handlers. A toolchain, similar to the Sancus toolchain, can be created to help a programmer write more secure code for IPE.

Implementation on OpenMSP430 To design countermeasures for the hardware security limitations of the IPE, it might be interesting to add IPE to Open-MSP430. From there, hardware-level countermeasures could be designed,

implemented, and tested. Limiting the execution range of the IPE section to an entry point solution like Sancus is the primary countermeasure to think about. From that stage, it is interesting to analyze and compare if IPE with a finer granularity where variables are not executable and code is not writable brings even stronger security guarantees.

TI's Documentation TI state themselves that MSP430 IPE is not perfect, but they do not mention in what manner it is not perfect. In this thesis, we indicate where IPE is not perfect. For a programmer, it is essential to understand the limitations of IPE. This is separated from the discussion of whether IPE could be used as part of a TEE. Even for code confidentiality, it is essential to understand these limitations.

A structured explanation in the TI manuals where they explicitly stated where IPE protects against and what the limitations are would make it more straightforward for a programmer to understand IPE. If TI designs a new version of IPE, it would be worth reducing these limitations, especially reducing the range from executable IPE to unprotected code.

Comparison MSP430 IPE and Sancus Comparing MSP430 IPE with Sancus is not completely honest to MSP430 IPE. As Sancus is an entire security architecture and IPE is a security feature to raise code confidentiality, they serve different purposes. However, looking at what can be done with security features from off-the-shelf MCUs like MSP430 IPE is from an academic point of view. Designing academic TEEs complementary to off-the-shelf MCU security features closes the gap between the academic design of a TEE and the commercial implementation of that academic TEE.

5.2 Contribution

In this thesis, we compared the TI MSP430 security features and, particularly, MSP430 IPE with Sancus, a security architecture that provides a trusted execution environment. With the results of this comparison, the key was extracted from the example code of MSP430 IPE.

The contributions of this master's thesis:

- We defined the security goals of MSP430 IPE isolation mechanism and MSP430 attestation based on different papers of TI.
- A detailed discussion of the isolation properties of MSP430 IPE is provided. The overall design was compared with the isolation mechanism of Sancus. Corner cases like the IVT and DMA are included as well. Alternatives of like MSP432 IP protected zones and applications like SIA were involved.
- We discussed methods on how code can be uploaded to an MSP430 MCU. The evolution from the BSL, to the crypto-bootloader, to applications that

implement their own custom BSL with attestation mechanisms. We compared this with the attestation Sancus provides.

- We pointed out the hardware and software limitations of MSP430 IPE. Based on these hardware limitations, three attack primitives were introduced. For each of these, several methods were experimentally introduced on how they can disturb the proper working of the code.
- We used a combination of the attacker primitives to attack the example code of MSP430 IPE in two manners. In the first method, the key was extracted out of the IPE section. In a second manner, the key was zeroed out.

Appendices

Appendix A

Technical Images MSP430FR5969

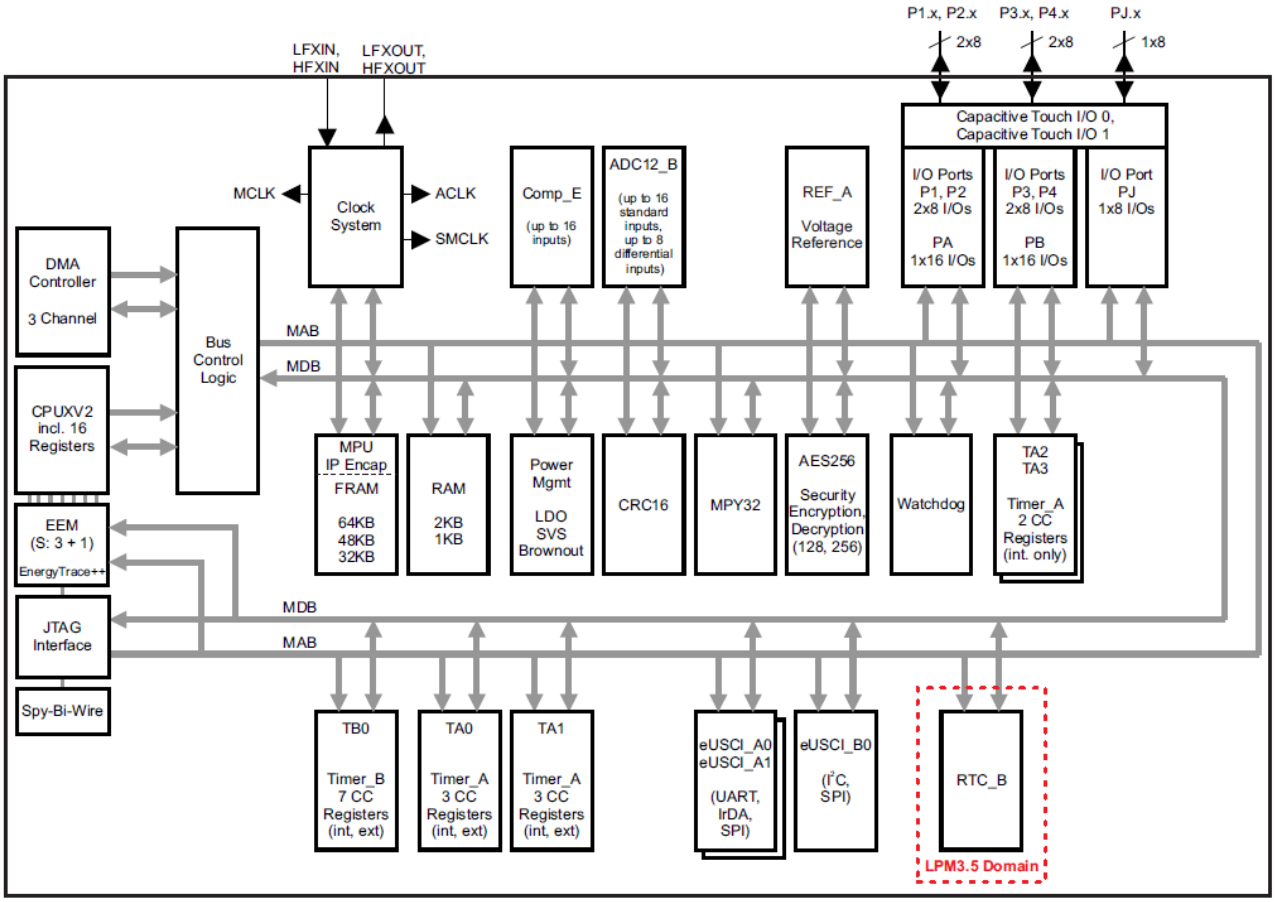

Figure A.1: Functional block diagram of the MSP430FR5969 microcontroller as represented in the manual [15]

.

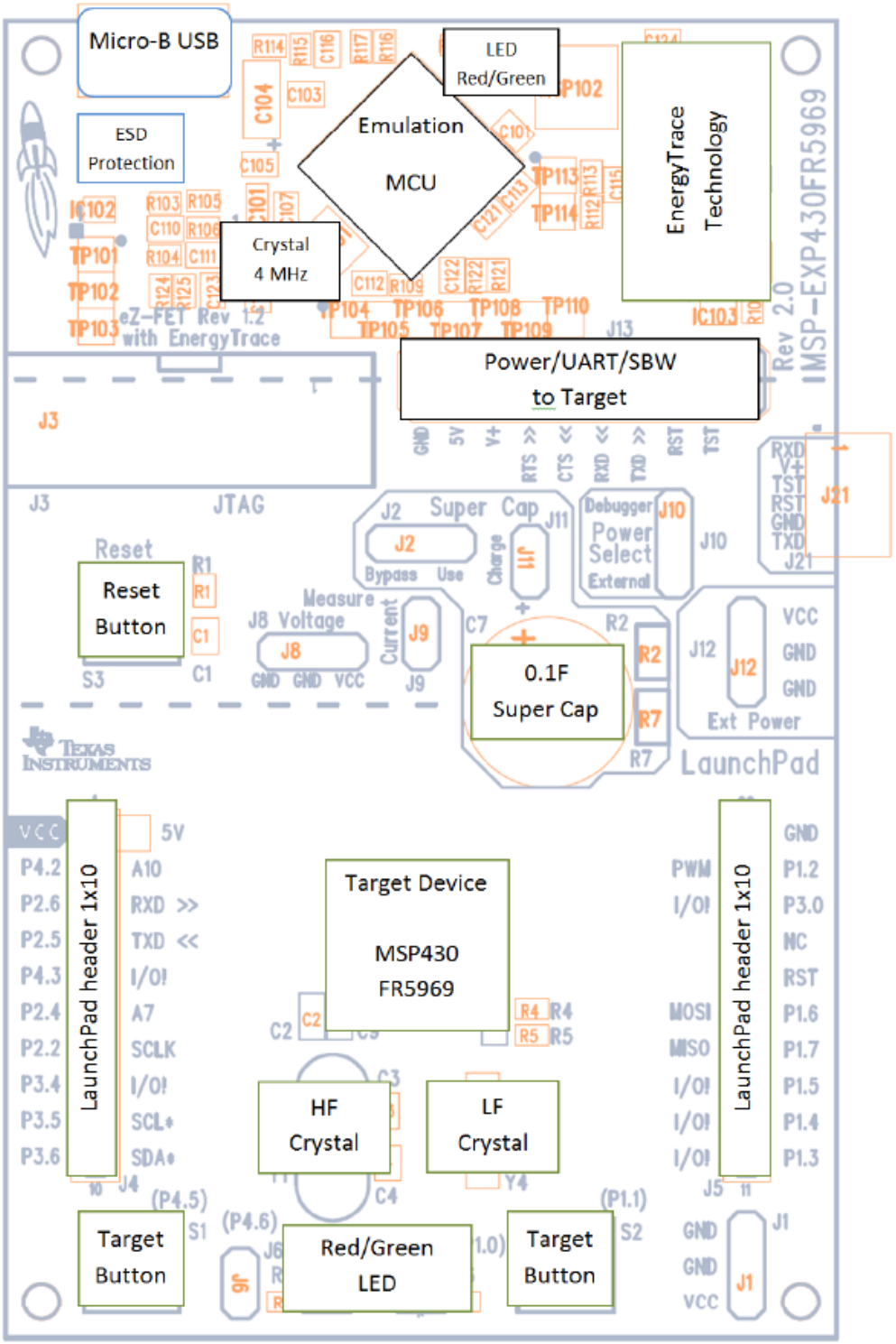

FIGURE A.2: Overview of the components of the MSP-EXP430FR5969 launchpad as represented in the user's guide [20].

Appendix B

TI Example Code IPE

The following code is provided by TI [37] and can be downloaded with http://www.ti.com/lit/zip/slaa685.

```
/* --COPYRIGHT--,BSD
 * Copyright (c) 2015, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with
 * or without modification, are permitted provided that the
 * following conditions are met:
 *
 * *  Redistributions of source code must retain the above
 *    copyright notice, this list of conditions and the
 *    following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the
 *    above copyright notice, this list of conditions
 *    and the following disclaimer in the documentation
 *    and/or other materials provided with the
 *    distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated
 *    nor the names of its contributors may be used to
 *    endorse or promote products derived from this
 *    software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 * CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
 * THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY
```

```
30  * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
31  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
32  * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
33  * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
34  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
35  * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
36  * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
37  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
38  * THE POSSIBILITY OF SUCH DAMAGE.
39  * --/COPYRIGHT--*/
40 /*************************************************************
41  *
42  * IPE_FR59xx.c
43  *
44  * Demonstrates a framework for using IP Encapsulation (IPE)
45  * on MSP430FR5xx/6xx devices. Shows how to place code in
46  * the IPE segments so that IPE setup can be performed by
47  * the tool built-in to CCS. It also demonstrates some
48  * IPE best  practices, like keeping all variables used
49  * by the IPE code also inside the IPE area, and clearing
50  * all hardware  registers used by the IPE code before
51  * exiting the IPE area. Several different types of data and
52  * code are stored to demonstrate different potential uses
53  * (variables, constants, functions).
54  *
55  * This code is meant to run on the MSP-EXP430FR5969
56  * launchpad, but could be adapted to run on other boards
57  * or devices that have IPE capabilities.
58  *
59  * The main code simply toggles a green LED roughly once per
60  * second (once for each iteration of the while(1) loop,
61  * which includes a delay of 500k cycles). The main while(1)
62  * loop also  calls IPE_encapsulatedBlink() from the IPE area
63  * once per  iteration. The IPE_encapsulatedBlink() function
64  * toggles the  red LED quickly, a number of times
65  * corresponding to the current value of
66  * IPE_encapsulatedCount stored in the IPE area.
67  * IPE_encapsulatedCount is incremented after each call to
68  * IPE_encapsulatedBlink, so there is one additional blink
69  * each  time, until there are 5 red LED blinks at which
70  * point it resets to 1.
71  *
72  * Make sure to use the IPE_FR59xx_load Debug configuration
73  * for loading new software. This is needed to erase the IPE
74  * area. To test the encapsulation preventing readout,
```

```
 * either use the "Hard reset" option while in the debug
 * mode, or power-cycle the device and then use the
 * IPE_FR59xx_test Debug configuration to debug without
 * re-loading the code. All variables and addresses inside
 * the IPE area should read 0x3FFF when encapsulation has
 * taken effect.
 *
 * NOTE: The first time loading a new project after having
 * this code present on the device with the IPE enabled, in
 * the new project Debug settings select "On connect, erase
 * main, information, and IP protected area" - otherwise a
 * verification error will occur because the project will
 * not be able to write to the previously protected area
 * of memory.
 *
 * The IPE area contains:
 * ----------------------
 * variables: IPE_i, IPE_encapsulatedCount
 * constants: IPE_encapsulatedKeys()
 * functions: IPE_encapsulatedInit(),
 *            IPE_encapsulatedBlink()
 * IPE structure is also included, but this is done
 * automatically by the IPE tool built into CCS.
 *
 * K. Pier
 * Texas Instruments Inc.
 * November 2015
 * Built with CCS v6.1.1 and IAR v6.40.1
 *
 ******************************************************************/
#include <msp430.h>
#include <stdint.h>

/*
 * IPE_encapsulatedCount is a variable inside the IPE area.
 * It is used by the IPE_encapsulatedBlink() function to
 * track the number of times to toggle the red LED
 */
#pragma DATA_SECTION(IPE_encapsulatedCount, ".ipe_vars")
uint8_t IPE_encapsulatedCount;

/*
 * IPE_i is a variable inside the IPE area.
 * It is used by the IPE_encapsulatedBlink() function as
 * a simple loop counter. This is used instead of a local
```

```
120  * variable, to keep it from being in RAM
121  * where it would be accessible
122  */
123 #pragma DATA_SECTION(IPE_i, ".ipe_vars")
124 uint8_t IPE_i;
125
126 /*
127  * IPE_encapsulatedKeys is a constant inside the IPE area.
128  * It demonstrates how to store keys and other consts inside
129  * the IPE area.
130  */
131 #pragma DATA_SECTION(IPE_encapsulatedKeys, ".ipe_const")
132 const uint16_t IPE_encapsulatedKeys[] =
133     {0x40A2, 0x0032, 0x4832, 0xCDEF,
134      0xAAAA, 0xBBBB, 0xCCCC, 0xDDDD};
135
136 /*
137  * otherFramVar demonstrates how to store a variable in
138  * FRAM but outside the IPE area.
139  * It is used simply to represent any other FRAM variables
140  * in a real application
141  */
142 #pragma PERSISTENT(otherFramVar)
143 uint8_t otherFramVar = 0;
144
145 /*
146  * myArray is an array stored in RAM.
147  * It is used simply to represent other RAM variables
148  * in a real application
149  */
150 volatile uint8_t myArray[10] =
151     {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
152
153 void IPE_encapsulatedInit(void);
154 void IPE_encapsulatedBlink(void);
155
156 /*
157  * main.c
158  */
159 void main(void) {
160     WDTCTL = WDTPW | WDTHOLD;   //Stop watchdog timer
161
162     /*
163      * Initialize the variables inside the encapsulated area
164      */
```

```
165     IPE_encapsulatedInit();
166
167     /*
168      * Initialize other variables used by the application
169      * that are stored FRAM
170      */
171     otherFramVar = 0;
172
173     /*
174      * Initialize GPIOs.
175      * Set P1.0 up for the green LED output.
176      * Unlock GPIOs for settings to take effect.
177      */
178     P1DIR |= BIT0;
179     PM5CTL0 &= ~LOCKLPM5;
180
181     while(1)
182     {
183         /*
184          * Code to represent operations happening
185          * on FRAM variables outside IPE
186          */
187         otherFramVar++;
188
189         /*
190          * Code to represent operations happening
191          * on RAM variables
192          */
193         myArray[0]++;
194
195         /*
196          * Toggle the green LED to show one iteration
197          * through the main while(1)
198          */
199         P1OUT ^= BIT0;
200
201         /*
202          * Delay to make the green LED toggle visible
203          */
204         //__delay_cycles(500000);
205
206         /*
207          * Call IPE_encapsulatedBlink(), a function
208          * stored in the IPE area.
209          */
```

```
210         IPE_encapsulatedBlink();
211
212         /*
213          * Code to represent operations happening on
214          * FRAM variables outside IPE
215          */
216         if(otherFramVar > 10)
217                 otherFramVar = 0;
218     }
219 }
220
221 /*
222  * IPE_encapsulatedInit
223  *
224  * Function stored in the IPE area.
225  * This function initializes any variables stored in
226  * the IPE area, because they cannot be initialized by
227  * normal C-startup code because that is outside the
228  * IPE area.
229  */
230 #pragma CODE_SECTION(IPE_encapsulatedInit, ".ipe")
231 void IPE_encapsulatedInit(void)
232 {
233         /*
234          * IPE_encapsulatedCount initialized to 1 for 1 blink
235      * on first execution of
236          * IPE_encapsulatedBlink()
237          */
238         IPE_encapsulatedCount = 1;
239 }
240
241 /*
242  * IPE_encapsulatedBlink
243  *
244  * Function stored in the IPE area
245  * This function represents the actual code that would
246  * be the proprietary code IP to be hidden in the
247  *encapsulated area.
248  *
249  * In this case, it simply:
250  * 1. Initializes the red LED
251  * 2. Initializes a timer module for the delay between
252  *    LED toggles
253  * 3. Toggles the LED for a number of times set by the
254  *    IPE_encapsulatedCount IPE variable.
```

```
255  * 4. Clears all module registers used (Timer, GPIO)
256  *    after the code finishes.
257  *
258  * It also demonstrates the use of a constant in the
259  * IPE area to load AES keys into the AES module, and
260  * clearing the AES module key at the end again so that
261  * the keys are not already loaded for use by any code
262  * outside IPE.
263  */
264 #pragma CODE_SECTION(IPE_encapsulatedBlink, ".ipe")
265 void IPE_encapsulatedBlink(void)
266 {
267         /*
268          * Initialize GPIOs used by IPE code.
269          * Set P4.6 up for the red LED output.
270      * Starts out low.
271          */
272     P4OUT &= ~BIT6;
273     P4DIR |= BIT6;
274
275     __asm("␣MOV.W␣#50,␣&0x4832");
276
277     /*
278      * Initialize the timer for setting the delay
279      * between red LED toggles
280      * 50000/~1000000 = 50ms delay (20 times per second)
281      */
282     TA0CCTL0 = CCIE;
283     TA0CCR0 = 50000;
284
285     /*
286      * Loading AES keys from IPE const section
287      */
288     for(IPE_i = 0; IPE_i < 8; IPE_i++)
289         AESAKEY = IPE_encapsulatedKeys[IPE_i];
290
291     /*
292      * Red LED toggles for the number of times stored in the
293      * IPE_encapsulatedCount variable
294      */
295     for(IPE_i = 0; IPE_i < IPE_encapsulatedCount; IPE_i++)
296     {
297         //Turn red LED on
298         P4OUT |= BIT6;
299         //Start timer
```

```
300         TA0CTL = TASSEL__SMCLK| MC__UP | TACLR;
301         //Sleep until timer int (50ms)
302         __bis_SR_register(LPM0_bits + GIE);
303         __no_operation();
304         //Turn red LED off
305         P4OUT &= ~BIT6;
306         //Sleep until timer int (50ms)
307         __bis_SR_register(LPM0_bits + GIE);
308         __no_operation();
309         //Stop timer
310         TA0CTL = 0;
311     }
312
313     /*
314      * Increment IP_encapsulatedCount to toggle 1
315      * more time on next call of
316      * IPE_encapsulatedBlink(), up to 5 times max
317      */
318     if(IPE_encapsulatedCount < 5)
319         IPE_encapsulatedCount++;
320     else
321         IPE_encapsulatedCount = 1;
322
323     /*
324      * Reset AES module so that keys and settings
325      * will be cleared, so it can't be used by any
326      * code outside of the IPE area
327      */
328     AESACTL0 = AESSWRST;
329
330     /*
331      * Clear all registers used (IPE best practice
332      * to conceal what IPE code does)
333      * Clear the timer registers and port registers
334      * in this case
335      */
336     TA0CCTL0 = 0;
337     TA0CCR0 = 0;
338     TA0CCR1 = 0;
339     TA0CCR2 = 0;
340     TA0CTL = 0;
341     TA0R = 0;
342     P4DIR &= ~BIT6;
343     P4OUT &= ~BIT6;
344
```

```
345     /*
346      * Clear General purpose CPU registers R4-R15
347      *
348      * Note: if passing parameters back to code outside
349      * IPE, you may need to preserve some CPU registers
350      * used for this. See www.ti.com/lit/pdf/slau132
351      * MSP430 C/C++ compiler guide section on How a
352      * Function Makes a Call and How a Called Function
353      * Responds for more information about the registers
354      * that the compiler uses for these parameters,
355      * R12-R15. Note that if too many arguments are passed,
356      * the stack will be used as well.
357      *
358      * In this case the function is declared void(void),
359      * so there is no issue.
360      */
361     __asm("␣mov.w␣#0,␣R4");
362     __asm("␣mov.w␣#0,␣R5");
363     __asm("␣mov.w␣#0,␣R6");
364     __asm("␣mov.w␣#0,␣R7");
365     __asm("␣mov.w␣#0,␣R8");
366     __asm("␣mov.w␣#0,␣R9");
367     __asm("␣mov.w␣#0,␣R10");
368     __asm("␣mov.w␣#0,␣R11");
369     __asm("␣mov.w␣#0,␣R12");
370     __asm("␣mov.w␣#0,␣R13");
371     __asm("␣mov.w␣#0,␣R14");
372     __asm("␣mov.w␣#0,␣R15");
373
374     /*
375      * At this point, it may also be desired to clear any
376      * RAM that was allocated in the course of the function,
377      * because RAM is not cleared on de-allocation. However,
378      * this is completely application-dependent and care
379      * would need to be taken not to corrupt the stack that
380      * is needed by the application for proper functioning.
381      * It may be better in cases like this to either:
382      * 1. Use no local variables, and therefore only use
383      *    variables in FRAM IPE area, to have less concern
384      *    (though there could still be some stack usage
385      *    leaving things in RAM after execution). This is
386      *    what is done in this example (see how even the loop
387      *    counter IPE_i is placed in IPE rather than a local
388      *    variable).
389      * 2. If there is real concern about what could be
```

```
390         *    placed in RAM, write the IPE routines in assembly
391         *    so that the user can completely control RAM and
392         *    stack usage.
393         */
394 }
395
396 /*
397  * TIMER0_A0_ISR
398  *
399  * Demonstrates placing an ISR in the IPE area. This ISR is
400  * used for the timer delay for the LED toggling in the IPE
401  * area, and simply wakes the device.
402  */
403 #pragma CODE_SECTION(TIMER0_A0_ISR, ".ipe:_isr")
404 #pragma vector=TIMER0_A0_VECTOR
405 __interrupt
406 void TIMER0_A0_ISR(void)
407 {
408     //Wake the device on ISR exit
409     __bic_SR_register_on_exit(LPM0_bits);
410     __no_operation();
411 }
```

Bibliography

[1] M. Bognár. Analyzing side-channel leakage in secure dma solutions, 2020.

[2] M. Bognár, J. Van Bulck, and F. Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. 2022.

[3] Y. Chen, Y. Zhang, Z. Wang, and T. Wei. Downgrade attack on trustzone, 2017.

[4] L. V. Davi. Code-reuse attacks and defenses. 2015.

[5] D. Dinu, A. S. Khrishnan, and P. Schaumont. Sia: Secure intermittent architecture for off-the-shelf resource-constrained microcontrollers. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 208–217, 2019.

[6] L. Giner, A. Kogler, C. Canella, M. Schwarz, and D. Gruss. Repurposing segmentation as a practical LVI-NULL mitigation in SGX. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022. USENIX Association.

[7] O. Girard. openmps430, 2017.

[8] O. Guillen, B. Nisarga, L. Reynoso, and R. Brederlow. Crypto-bootloader - secure in-field firmware updates for ultra-low power mcus. URL: https://www.ti.com.cn/cn/lit/wp/slay041/slay041.pdf, last checked on 2022-06-07, September 2015.

[9] R. Hariharan. *Software IP Protection on MSP432P4xx Microcontrollers*, November 2016.

[10] T. Instruemnts. Closing the security gap with ti's msp430™ fram-based microcontrollers. URL: https://www.ti.com/lit/wp/slay035/slay035.pdf, last checked on 2022-06-07, July 2010, revised May 2021.

[11] T. Instruments. Msp430 microcontroller family. URL: https://www.ti.com/sc/data/msp/databook/chp1.pdf, last checked on 2022-06-07.

[12] T. Instruments. Msp430 optimizing c/c++ compiler v21.6.0.lts. URL: https://www.ti.com/lit/ug/slau132y/slau132y.pdf, last checked on 2022-06-07, September 2004, revised June 2021.

[13] T. Instruments. Fram - new generation of non-volatile memory. URL: https://www.ti.com/lit/ml/szzt014a/szzt014a.pdf, last checked on 2022-06-07, 2009.

[14] T. Instruments. Msp430fr58xx, msp430fr59xx, and msp430fr6xx family user's guide. URL: https://www.ti.com/lit/ug/slau367p/slau367p.pdf, last checked on 2022-06-07, October 2012, revised April 2020.

[15] T. Instruments. Msp430fr596x, msp430fr594x mixed-signal microcontrollers. URL: https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf, last checked on 2022-06-07, October 2012, revised August 2018.

[16] T. Instruments. Errata msp430fr5969 microcontroller. URL: https://www.ti.com/lit/er/slaz473ag/slaz473ag.pdf, last checked on 2022-06-07, 2012, revised August 2021.

[17] T. Instruments. Direct memory access (dma) controller module. URL: https://www.ti.com/lit/ug/slau395f/slau395f.pdf, last checked on 2022-06-07, August 2012, revised March 2018.

[18] T. Instruments. Msp430™ fram technology - how to and best practices. URL: https://www.ti.com/lit/an/slaa628b/slaa628b.pdf, last checked on 2022-06-07, June 2014, August 2021.

[19] T. Instruments. Msp430™ fram devices bootloader (bsl). URL: https://www.ti.com/lit/ug/slau550aa/slau550aa.pdf, last checked on 2022-06-07, January 2014, revised February 2021.

[20] T. Instruments. Msp430fr5969 launchpad™ development kit (msp-exp430fr5969). URL: https://www.ti.com/lit/ug/slau535b/slau535b.pdf, last checked on 2022-06-07, February 2014, revised July 2015.

[21] T. Instruments. Secure in-field firmware updates for msp mcus. URL: https://www.ti.com/lit/an/slaa682/slaa682.pdf, last checked on 2022-06-07, November 2015.

[22] T. Instruments. Cmsp430 gcc toolchain. URL: https://www.ti.com/lit/ug/slau646f/slau646f.pdf, last checked on 2022-06-07, September 2015, revised June 2020.

[23] T. Instruments. How planetary resources uses ti msp43™ mcus to discover earth's resource from space. URL: https://www.ti.com/lit/ml/slat158/slat158.pdf, last checked on 2022-06-07, October 2016.

[24] T. Instruments. Understanding security features for msp430 microcontrollers. URL: https://www.ti.com/lit/ml/swpb018/swpb018.pdf, last checked on 2022-06-07, 2017.

[25] A. Jajoo. A study on the morris worm, 12 2021.

[26] A. S. Khrishnan, C. Suslowicz, and P. Schaumont. Secure and stateful power transitions in embedded systems. In *Journal of Hardware and Systems Security*, pages 263–276, December 2020.

[27] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[28] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel. Intermittent Computing: Challenges and Opportunities. In B. S. Lerner, R. Bodík, and S. Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[29] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.

[30] J. T. Mühlberg, S. Cleemput, M. A. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. An implementation of a high assurance smart meter using protected module architectures. In S. Foresti and J. Lopez, editors, *Information Security Theory and Practice*, pages 53–69, Cham, 2016. Springer International Publishing.

[31] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.

[32] B. Nisarga and E. Peeters. System-level tamper protection using msp mcus. URL: https://www.ti.com/lit/an/slaa715/slaa715.pdf, last checked on 2022-06-07, August 2016.

[33] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3), jul 2017.

[34] J. Noorman, R. Strackx, F. Piessens, B. Preneel, and I. Verbauwhede. Protected software module architectures, 2013.

[35] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. VRASED: A verified Hardware/Software Co-Design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, Santa Clara, CA, Aug. 2019. USENIX Association.

[36] I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik. On the toctou problem in remote attestation, 2020.

[37] K. Pier. Msp code protection features. URL: http://www.ti.com/lit/an/slaa685/slaa685.pdf, last checked on 2022-06-07, December 2015.

[38] F. Piessens and I. Verbauwhede. *"Software security: Vulnerabilities and countermeasures for two attacker models." Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, October 2016.

[39] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. 51(6), jan 2019.

[40] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security Privacy*, 10(6):84–87, 2012.

[41] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015.

[42] J. Schang. Enabling secure portable medical devices with ti's msp430™ mcu and wireless technologies. 2013.

[43] S. Seminara. Dma support for the sancus architecture. Master's thesis, Politecnico di Torino, 2019.

[44] R. Strackx. *Security Primitives for Protected-Module Architectures Based on Program-Counter-Based Memory Access Control*. KU Leuven.Faculteit ingenieurswetenschappen, Leuven, 2014.

[45] C. Team. Cmsp430 gcc toolchain. URL: https://clang.llvm.org/docs/UsersManual.html, last checked on 2022-06-07, 2022.

[46] P. Thanigai. Msp430™ programming with the jtag interface. URL: https://www.ti.com/lit/ug/slau320aj/slau320aj.pdf, last checked on 2022-06-07, July 2010.

[47] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In R. Sommer, D. Balzarotti, and G. Maier, editors, *Recent Advances in Intrusion Detection*, pages 121–141, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[48] J. Van Bulck, J. T. Mühlberg, and F. Piessens. Vulcan: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 225–237, New York, NY, USA, 2017. Association for Computing Machinery.

[49] J. Van Bulck, F. Piessens, and R. Strackx. *Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic*, 2018.

[50] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. Hybrid dataflow/von-neumann architectures. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1489–1509, 2014.

[51] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Cryptology ePrint Archive, Paper 2005/388, 2005. https://eprint.iacr.org/2005/388.