Fachbereich
Elektrotechnik und Informatik

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Bachelor Thesis

Design and Evaluation of a Tiny Instruction Emulation for Security Enhanced Embedded Systems

Guhao Huang

Date of handing out: December 20th 2017
Date of handing in: September 14th 2017

(Prof. Dr. rer. nat. Oliver Stecklina)
Board of Examinations

Task description

The ubiquitous use of embedded computer systems is tightly coupled with an increasing interest of security technology on tiny systems. Off-the-shelve tiny microcontrollers are usually not equipped with sufficient security mechanisms. The restricted resources of these components make an implementation of modern security mechanism a challenging task.
In this bachelor thesis the design of a tailor-made virtualization by implementing a tiny hypervisor should be proved. An enforcement of security mechanisms is possible by an emulation of, a subset of instructions. During the bachelor thesis the instruction emulation of a tiny hypervisor has to be implemented for an MSP430 microcontroller. Furthermore, the overhead of instruction emulation has to be measured to give a proofed database of further performance analysis of typical applications.

Fachbereich Elektrotechnik und Informatik
Dekanat - Prüfungsausschuss

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Declaration of the Candidate

I, the undersigned, hereby declare that the work contained in this thesis is my own original work, and has not previously in its entirety or in part been submitted at any university for a degree.

Only the sources cited in the document have been used in this draft. Parts that are direct quotes or paraphrases are identified as such.

I agree that my work is published, in particular that the work is presented to third parties for inspection or copies of the work can be passed on to third parties.

Lübeck, December 16th 2017

. . . . . .

(signature)

Contents

1 Introduction

1.1 Motivation

The embedded system as a revolutionary portable solution for many technology, has more and more functions and tasks. From primitive applications like boolean logical controller, to the modern television, mobile phone and other system with multi modules and networks, the complexity of embedded system grows. More and more system failures happen at runtime. The problems are various. Some of the problems are caused by the system itself, some are caused the attacker. But they have a common reason, the security defect. For example, buffer over flow and other problems may cause the system run in a unstable status. The more complex the system is, the more security is demanded for the system. To ensure that the system operation is stable , the security is the key fact. The embedded system usually equipped with a very limited system resources. To design such a system, security and efficiency should be considered.
To implement the security enhanced system, instruction emulation is a crucial part. Emulation technology is a basement of security enhanced embedded system. Emulation offers a security enhanced interface for the software. The select of a good emulation technology is also a important part to design a security enhanced embedded system.
Before the system is designed, the knowledge of the platform is also important. For embedded systems, the platform most rapidly is a bare hardware. The development is built rapidly with assemblers.

1.2 Goal

The goal of this thesis is to develop a instruction emulation for a security enhanced embedded system. The kernel is a miniaturized component that offers a secure interface for the software. The kernel implements an instruction emulation. The two main function of Hypervisor is to verify the write/read/call operation at runtime. The second is to emulate the instructions which is emulated by the security enhanced interface. At the same time, the efficiency of the kernel must be ensured. In this thesis, the research on MSP430 instruction set is done first. To understand the security defect and platform is also a goal of the thesis.

1.3 Organization

The rest of the document is organized in the following structure. Chapter 2 is the introduction of the MSP430 micro controller. Chapter 3 is the principle of embedded system security and the methodology of building a security enhanced embedded system. Chapter 4 introduces emulation technologies and the method applied in the concrete implementation. These three chapters are the background knowledges and methodology for the implementation. In Chapter 5, the implementation details including the structure and components are given. Chapter 6 includes the evaluation. To do the evaluation and development, the environment for development and test are described in the first part of Chapter 6, and the evaluation is shown in Section 6.2 and 6.3.

2 MSP430 Micro Controllor

This chapter is an introduction to the MSP430 controller which is a product of Texas Instruments.

2.1 Hardware

The MSP430 is a Micro Controller that is famous for its ultra-low power. And this controller is a whole system including RAM, ROM, Peripherals and an CPU. This CPU is provided with a 16-bit and RISC (Reduced Instruction Set Computer) architecture. MSP430 has a von-Neumann architecture. Von-Neumann architecture restore data and the program in a same form. It is good for various sensors and modules.

2.1.1 Clocks

This controller has a flexible clock system in order to promote its super energy saving. In fact, it has 2 common clocks, ACLK at 32kHz, SMCLK and MCLK at 1mHz. SMCLK is for the subsystems.

Low-frequency auxiliary clock = Ultra-low-power stand-by mode;
High-speed master clock = High performance signal processing [Ins13].

2.1.2 Address Space

MSP430 has a continuous abstract address apace, which means ROM, RAM and resources are integrated in a sole address space. To MSP430x architecture, a word is 20 bits long, so that the space is from 0h-FFFFFh which is as large as 1MB. To MSP430 architecture, a word is 16 bits long, so that the space is from 0h-FFFFh.
The Address Space is a one-one map from the address to its value restored. The memory obtain an 8 bits length for each address, which is called a byte. The peripheral modules and the interrupt vectors are obligated in the space. The memory of MSP430 is non-paged. And the instruction set can address 1MB memory without paging. There are also a Binary Mode, which will only address the 8 bits of a memory space. The other is word mode, which will address 16 bits or 20 bits starting from the address. As shown in the 2.3, it is a memory space of a MSP430 Micro Controller. 1FFFFh-0FFFFh is the flash/ROM in the 20-bit architecture, extended by the 4 bits difference. 0FFFFh-0FFE0h is the Interrupt

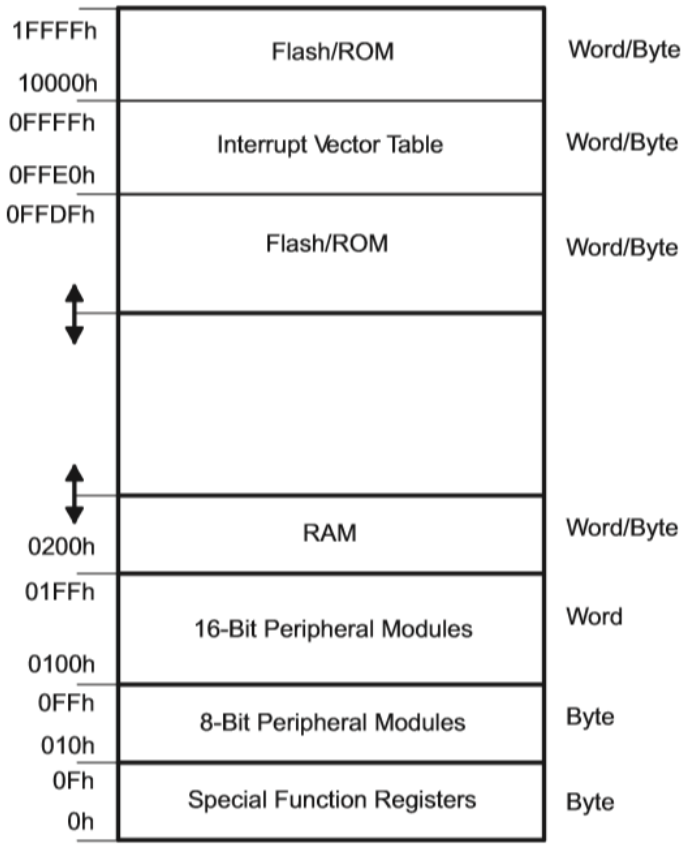

Figure 2.1: Memory Space [Ins13]

Vector Space. 0FFDFh-0200h is RAM, which can surmount on the ROM 01FF-010h is the Peripheral Module space. 0Fh-0h is Special Function Registers.

2.1.3 CPU

The CPU registers are discussed here. The CPU registers are the high-speed caches. From R0 to R15, there are 16 words used as the CPU registers.

- PC(R0) points to the next instruction to be executed. PC is short for Program Counter. After an instruction is executed, the PC is added by certain value decided by the instruction that is executed.
- SP(R1), Stack Pointer is used as the address of stack. Stack is restoring data after interrupts and calls or used as a temporary store. Stack is a part of the memory space, and Stack Pointer restore the starting address of the whole stack.
- SR(R2), Status Register has 16 bits of status. From the 8 bit to 0 bit are V, SCG1, SCG0, OSCOFF, CPUOFF, GIE, N, Z and C in proper order. V is the overflow bit.

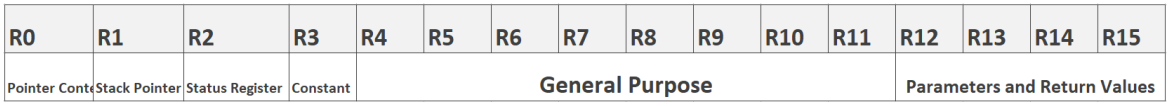

Figure 2.2: CPU Registers

SCG1 and SCG0 are the clock setting bits. N is a flag bit of negative value is generated. Z is the flag of zero. The bits are automatically set or reset when the a instruction cycle finished. GIE enables the maskable interrupt when set.

- R3 and R2 are the constant register which generates the constant values when used in certain addressing mode. They can be used to create some value easily.
- R4 to R15 are general purpose register. C compiler usually restores the return value or parameters of function in R12 to R15.

2.1.4 Stack

Stack is not a hardware, but is a part of the RAM which is pointed by the Stack Pointer, starting from the address Stack Pointer restoring. When stack increases, the SP subtracted by 2. The top of stack is the 0(SP). The offset to the top is #N. An address in the stack is presented as #N(SP), N $\geq$ 0.

2.2 Instruction Set

The MSP430 instruction set consists of 27 core instructions and 24 emulated instructions. [Ins13] The instructions divide into Dual-operand, Single-Operand and Jump instructions. Here we define the fields as the User's Guild by TI.
MSP430x architecture has instruction set that is additional with instruction for the 20-bit word operation. These instructions have an 'A' ending comparing to 16-bit instruction, for example 'CALLA', 'MOVA' and 'SUBA'. These instructions is officially called CPUX or MSP430 extended.

Field	**Definition**
Op-Code	The op-code bits is the classifier of different instruction.
As	The addressing bits responsible for the addressing mode used for the source (src)
S-reg	The working register used for the source (src)
Ad	The addressing bits responsible for the addressing mode used for the destination (dst)
D-reg	The working register used for the destination (dst)
B/W	Byte or word operation: 0: word operation 1: byte operation

Table 2.1: Definition of instruction fields [Ins13]

These fields form an instruction. In the machine, they are all in binary. The core instructions are 16 bits, 32 bits or 48 bits long. The length of an instruction depends on the addressing mode and the operation it does. Next, the details of the instructions will be discussed.
The source operand is the first of the two operands. It represents the address, the register or the value. The destination operand represents the address or the register that the data pass to.

2.2.1 Instruction Structures

- The structure of double operand instructions is

  Op-code ***S-Reg*** ***Ad*** ***B/W*** ***As*** ***D-Reg***

  The name of these instructions are MOV, SUB, ADD, ADDC, SUB, SUBC, CMP, DADD, BIT, BIC, XOR and AND. These instructions are involving two registers or values.

- The structure of single operand instructions is

  Op-code ***B/W*** ***Ad*** ***D/S-Reg***

  The name of these instructions are RRC, RRA, PUSH, SWPB, CALL, RETI and SXT. These instructions are involving one registers or address.

- The structure of jump instructions is

  Op-code ***C*** ***10-Bit PC Offset***

  The name of these instructions are JZ, JNE, JC, JNC, JN, JGE, JL and JMP. These instructions are used to jump to another location of program. The 10-Bit PC offset offer a jump range from -511 to +512 words relative to the current PC value.

All instructions in MSP430 are organized in the three ways. But in some conditions, the length will be more than those above. That is because in different addressing mode there are three or more operands in one instruction, such as *MOV R5, 2(R1)*. The additional Operand (usually a value) is put in the next 16 bits of this word where the instruction locates. How these works will be shown in the next sections.

2.2.2 Binary Representation

The form that instructions used in the machine is binary code. CPU refers to Program Counter to look for the address of the next instruction from the memory. To understand the machine code better, it must be clear that all the fields of instructions are one-one mapped to specific binaries. Binaries of the instruction are expressed in the form of hexadecimal number. Map between the binary to fields shown in 2.1 is also defined in

the instruction set. The instruction in the memory (in the form of binary) is read by the machine. First, we have the Op-codes.

	000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
0xxx																
4xxx																
8xxx																
Cxxx																
1xxx	RRC	RRC.B	SWPB		RRA	RRA.B	SXT		PUSH	PUSH.B	CALL		RETI			
14xx																
18xx																
1Cxx																
20xx	JNE/JNZ															
24xx	JEQ/JZ															
28xx	JNC															
2Cxx	JC															
30xx	JN															
34xx	JGE															
38xx	JL															
3Cxx	JMP															
4xxx	MOV, MOV.B															
5xxx	ADD, ADD.B															
6xxx	ADDC, ADDC.B															
7xxx	SUBC, SUBC.B															
8xxx	SUB, SUB.B															
9xxx	CMP, CMP.B															
Axxx	DADD, DADD.B															
Bxxx	BIT, BIT.B															
Cxxx	BIC, BIC.B															
Dxxx	BIS, BIS.B															
Exxx	XOR, XOR.B															
Fxxx	AND, AND.B															

Figure 2.3: Core Instruction Map [Ins13]

As shown in the 2.3, the op-code fields in the form of hexadecimal number are classified. For different instructions, the op-codes have different length. The column shows the last 12 bits of the first word, and the rows shows the first 8 bits of the instruction. The of op-code is define by adding the row and column. In 2.4, the first word of all the three instruction is displayed. The detail of bits in the fields of the 2.1 demonstrated.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Double Operand Instruction	Op-Code				S-Reg				Ad	B/W	As		D-Reg			
Single Operand Instruction	Op-Code									B/W	As		D-Reg			
Jump Instruction	Op-Code			C					10-bit PC offset							

Figure 2.4: Binary Code Structures

With 2.3 we can analyze easily how the Op-code maps to the binary code. Take care that here are some hexadecimal numbers in the mapping. For example, 'MOV' is mapped to 4xxx, which means that the first 4 bit of the word is always 0100. There is some difference between

Single Operand Instructions and Double Operand Instructions. 'CALL' is mapped to 1280, which means that the first 9 bits of the instruction is 000100101, so that the final hexadecimal number for the binary code **'CALL @R4'** should be 0x12a4h(0001001011000100).
The Op-Code decides the operation of a instruction. And the addressing mode decide the operand and length of the an instruction.

2.2.3 Addressing Mode

There are seven addressing modes in MSP430. They are Register Mode, Indexed Mode, Symbolic Mode, Absolute Mode, Indirect Register Mode, Indirect Autoincrement and Immediate Mode. These modes are the key fact defining a instruction, which will be introduced as follow.

- The Register Mode, which deals with directly the data in the registers, is the simplest addressing mode. For example, in **'MOV R4, R5'** the Source and Destination works as register when and only when the As is 00, and Ad is 0. The register mode is both available for source register and destination register.

- The Indexed Mode, the Symbolic Mode and the Absolute Mode are using the same As/Ad bits. For the source, the As is '01'. For the destination register the Ad is '1'. For example, in the Indexed mode, **'MOV 0x0004(SP),R12'** is '411C 0004' in hexadecimal number. In this case, the source is in indexed mode. 4(SP) is representing the address in the memory. But it is also possible that destination register are using index mode. The Ad bit is 1. For example, the hexadecimal code of **'MOV R5,6(R6)'** is '4586 0006'. As shown in 2.4, the 8th-5th bit is the As/Ad bits and B/W. Here is 8 in hexadecimal, which is 1000 in binary. It's also possible to have an As/Ad like 01/1. The Index Mode is available both for the source register and destination register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15-0	15-0
MOV				0x0004(R6)								R12					
0	1	0	0	0	1	1	0	0	0	0	1	1	1	0	0	0x0004h	
Op-Code				S-Reg				Ad	B/W	As		D-Reg				NextAddress	Next2Address

Figure 2.5: Binary Instruction Sample

- The Indexed Mode and the Symbolic Mode have the same As/Ad bits, but the register is R0(PC). For example, **'MOV 2(PC), 3(PC)'**.

- The Absolute Mode is more different than the other two. For example, **'MOV &2344, &F27F'** has a similar hexadecimal code '4292 2344 F27F'. Actually, in Absolute Mode, the register involved in the operation is always R2, which means that

(#N)R2 represents an absolute address &N. The index mode, absolute mode and the symbolic mode are both available for the source register and the destination register.

- The Indirect Register Mode, as its name shows, uses the value in the register as a pointer to an address. For example, **'MOV @R10, 0(R12)'** is '4AEC 0000'. The As bits 10. Here R10 does not represent the value in the R10 register but address R10. The Indirect Register Mode is only available for the source register.

- Indirect Autoincrement Mode is similar to Indirect Register Mode. For example, **'MOV @R10+,0(R12)'** is '4ABC 0000', the difference is that R10 will be increased by 1 automatically (by 2 for a word operation). The As bits are 11. It is only available for the source register.

- The last one is The Immediate Mode. For example, if the current PC points to &FF16 , **'MOV #45, &10A8'** is '40B0 0045 1192' which is **'MOV @PC+, #1192(PC), #0045, #offset'** in another form. 1192 is the offset from current PC to the destination address. FF16 + 1192 = 10A8. The As bits are 11. When using a Immediate Mode, the source register is a flag. The operand is in the later 2 word and the destination register.

2.3 Example Instructions

In this section some typical instructions are introduced.
The instructions are executed in several cycles, which means that the operation is finished in several clock periods. The cycles of an instruction are give by its addressing mode. For example, the RETI (Return from Interrupt) has 5 cycles. The MOV in Register Mode only has one cycle.
Instructions consist of sub-operations. And those are the smallest units of the CPU operation. But the multi-cycle instructions are in fact inseparable. They must be executed entirely. The length of an instruction depends on the operation type and the addressing mode.

2.3.1 MOV

MOV is a double operand instruction. The register mode of MOV only required one cycle to be executed. The other mode such as Indexed mode required two cycles to reach the address, for both the source and destination. Then the source restored to a temporary address. Finally, move the data from restored source to the destination. So the Indexed Mode for both source and register required 6 cycles to execute. The Indirect Register Mode for source and Indexed Mode for the destination requires 5 cycles.

When the register in a MOV instruction is not Register Mode, the instruction need more operands to describe the source address. As a result, MOV instruction could be 1,2 or 3 word long. The other double operand instruction is in a same structure and addressing mode as MOV instruction.

2.3.2 PUSH/POP

POP and PUSH are instruction controlling stack whose cycle number varying from 3 to 5. POP is taking word from stack to destination location.

1. **@SP → Temp**
2. **SP+2 → SP**
3. **Temp → dst**

First the stack top is restored in temp, then the stack pointer is added by 2, which reduced the stack space by one word. Finally move the restored data to the destination.
Contrary to POP, PUSH is restoring word to the top of the stack.

1. **SP-2 → SP**
2. **Src → @SP**

First, the stack is increased by one, then the data from source is restored into it.
For a single-operand instruction like POP/PUSH, it could be 1 or 2 words long, which depends oh the addressing mode of the only register.

2.3.3 CALL

There are 4 or 5 cycles in the procedure of CALL instruction.

1. **Dst → Tmp**
2. **SP-2 → SP**
3. **PC → @SP**
4. **Tmp → PC**

First, the Destination address is restored in a temporary address, then the Stack Pointer are reduced by 2 and the next instruction address is restored to the pointed address. Finally, the Pointer Counter change to the restored destination which will guide the program jump to another location in a memory. The CALL instruction replaces the order from executing sequentially. The restored PC in stack enable the program to return to the original position just after the CALL instruction. CALL is a single operand instruction. When the addressing mode of source is Register Mode or Indirect Register Mode, CALL instruction requires 4 cycles.
CALL is a single-operand instruction. It could be 1 or 2 words long.

2.3.4 RET

The RET instruction is to let the program return from sub-branches. It is in fact a MOV instruction. It move the restored PC from the stack into PC (R1). It is frequently seen when ending a function call and return.

1. **@SP → PC**
2. **SP+2 → SP**

What is worth paying attention to is that it is finished in an entire instruction. The assembler is **'MOV @SP+, PC'**, which is in Indirect Autoincrement Mode. And it is finished in 5 cycles. RET is usually matched after a CALL used. Normally, the program call a function and jump to some entrance, and the return is the exit of this function call.

3 Security of Embedded Systems

In this chapter, the security risks of embedded systems and the solutions adopted to solve the problems are expounded. Computer Security is a comprehensive task in information technology. It has three aims:

- To pretend theft of or damage to the hardware.
- To prevent theft of or damage to the information.
- To prevent disruption of service. [Gas88]

As to the embedded systems, they are also computer systems. But comparing to the large-scale systems, embedded systems are more isolated and function purposed, and they are much more less computing and memory capacity. To improve the computer security in embedded systems, their characteristics shall be sufficiently regarded.

3.1 Security and Safety

As to current software development procedure, many of them are done by higher-level language. The function and even Object-Oriented Class became the basic unit of a program. The developer can organize the usage of addressing space for variables. 'The higher your language, the farther from the hardware.' This thought caused other problems, for example leaks and bugs. The program may seems to be logically perfect in higher-level language, but there may be some bugs in the views of assembler. Compiler and frameworks organize the instructions automatically. The program may run in a status that likely has buffer overflow or instruction disorder. To create a platform avoiding these problems is an mission of Computer Safety.
But safety is not enough, nowadays, there are many forces trying to influence the computer systems to achieve their own goal, even though in a safe system. Computer security is a domain preventing and treating the subjective trial that is a thread for computer system.

3.2 Risks

Current universal computer is mainly Von Neumann Architecture which uses a shared memory space for data and instructions.

The risks and bugs are significantly common in embedded system development. There are two main method of attacker take use of the risks is to inject codes or to use existing code. To a high-level language programmer, the software is written in a view ignoring hardware. Some regulations are added in the compilers and other conventions. For example, the stack length, the return address of a function. It's also a possibility that the attacker overflow the key memory space by misuse some code. Or get the access to some system code. For example, the PlayStation crackers that install some software which can change the operating system. Some program even have bugs that will change the data without any attackers.
Some risks may caused by the people who can change the firmware. The influence may caused by changing system itself. For example, tampering attack, WSN module misuse, etc.

3.2.1 Basic Manipulations

- **Stack** When a program writes data into stack out of its frame, the stack buffer overflowed, which will cause the data or instructions to be overwritten in the memory space. Stack is a fixed memory space by program but not by the machine. As discussed in the Chapter 1, stack is increased by decreasing the stack pointer. The structure of a stack is shown in Figure 3.1. Decreasing the stack pointer, and making too much data into the stack is called stack smashing.[Ste16]

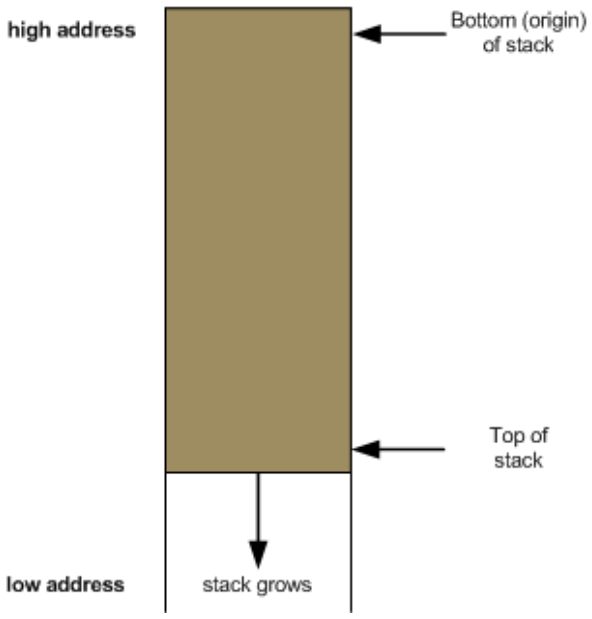

Figure 3.1: Stack [Ben11]

- **Function Call**

  Function pointer is a variable that containing the address of the function entrance. By constructing a buffer near the variable. The buffer can be overflowed onto the pointer,

which causing the value change of the pointer. When the pointer is called, the program will jump to somewhere the attacker wants to.

As shown in Figure 3.2, the stack changes when a function is called. The local variables and other storage is added into the stack and these depends on the language feature. The return address is also restored in the stack when a function is called. When the attacker can write a program controlling the stack, the stack can be used to control the return address of a function. The programmer can call any position in the middle of a function that can be take use of. The programmer can even call multi functions to construct an instruction series. The ROP (Return Oriented Programming) can also be applied in a Harvard Architecture that the instructions are separated with data as long as the stack can be controlled. [Ste16]

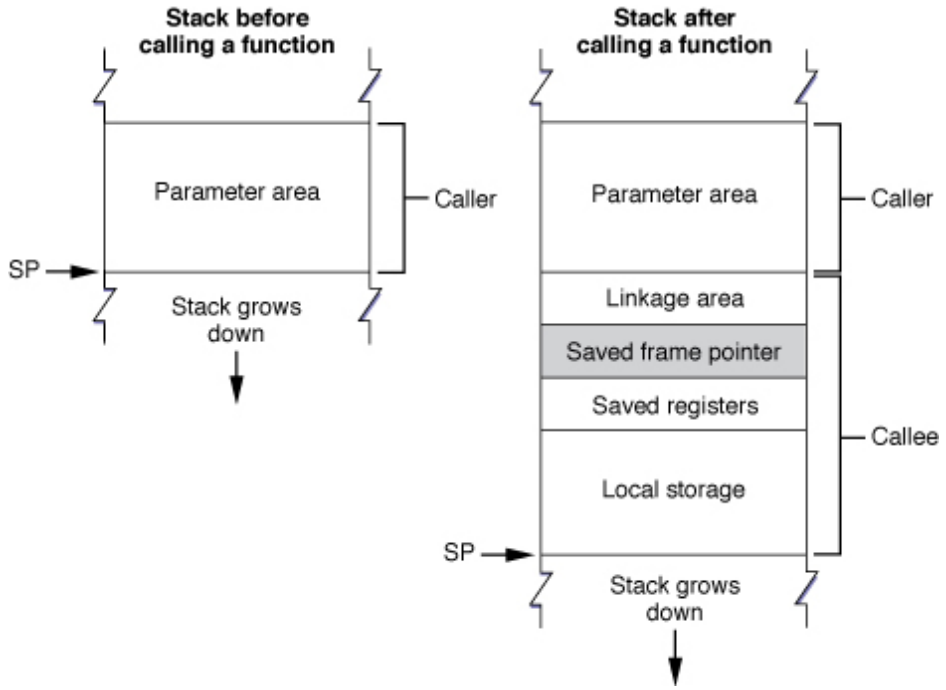

Figure 3.2: Stack change when a function calls [Inc10]

3.2.2 Problems in Multi-Module System

Some of the embedded systems have many modules. To these more complex systems, the developer would choose framework or other tools for development, which causes new problem: the compilers may not listen to the programmer. The isolation between modules is significantly important because the insecure compiler may create some illegal instructions that read/write data out of the module. In more complex cases, the system may have system core or private databases. These areas must be protected sufficient and deny any illegal

access. Normal programming language compiler would make big problem in run time. Some of the function may clean the database, some of them may write in the system core. If the system is user-oriented, the system detail should be hidden completely except for the user interface.

3.3 Goal

As illustrated in Section 3.2, the secure difficulty that the embedded system developers meet with is both technical and structural. The security problems are caused both by the techniques that the user are using and the basic structure of the embedded system. Facing these problems, some goal to solve the risks is raised.

- Small-Scale
- Avoid Vulnerabilities of Insecure Programming Language
- Ensure Module Security at Runtime

The first goal is small scale. The resources of embedded system is very limited. It is impossible to deploy a 100 thousand-line operating system in them. Large operating systems implementing a restrict security mechanism will be impossible for limited security feature in embedded system. For a common embedded system has no hardware memory protection and privilege level, small scale software solution is a good solution. So that the scale of any additional kernel software is very crucial.
To a programmer using higher level language such as C. The normal compiler and organization is not suitable any more. A new framework with secure kernel is a solution for it, which means that ensure the language safety.
The system resources and software should be protected by the system kernel, ensuring the system will not be damaged or misused by outside forces. Otherwise, the goal is to protect the system's security by a software mechanism but not when the attacker have access to the hardware such as tampering.

3.4 Solutions

There are many ways to maintain the computer security. For example, memory separation, access control, software-based memory protection. These solutions are tightly linked to operating systems. The main thought of these solutions is to build a layer between application software and hardware. The solution is letting the risks of security blocked by the software detection and resource isolation among tasks.

3.4.1 Access Control

In access control, the resource and service are divided into parts. The parts of the resources and services are called objects. The user of the subjects which can be a function, a real user or a class is called a subject. The subjects of the resource and service is controlled by verification of the access to the objects, which will authorize the legal read/write operation to certain objects only. One key method is access matrix that is a table whose columns are the subjects and rows are the objects. The access matrix is a discipline of all objects and subjects and their relationships. Technically, there are access control lists which takes a column of access matrix and capacities which takes a row instead. The former one can be joint with the objects, and latter one can be joint with subjects. There are also many type of access control in computer security, for example, RBAC (Role Based Access control) and MAC (Mandatory Access Control). [Ste16]

3.4.2 Memory Separation

Memory separation aims to divide memory into parts and restrict the access to these parts. For example, the operating system will restrict the access to write in the memory space of operating system.

- Protection Rings is also named Hierarchical protection domains. It's also an access control to the memory space. The core rings are in higher privilege level for example only kernels of operating system, then the outer rings are drivers and critical software, and the edge rings are normal data and applications.
- Segmentation and Paging are separating memory space into parts. The segmentation separated as the software parts. The size is allocated variably which causes space wasting. Paging is separating the memory into fixed sizes. They both address by segment/page offset method, so that the address is in fact a virtual address. The mapping to physical address to virtual address is supported by a hardware, MMU (Memory Managing Unit).

3.4.3 Software-Based Memory Protection

In embedded system, the possibilities is higher for the system lacking MMU. There are some software based methods to do the task of memory separation. SFI (Software Based Fault Isolation), CFI (Control Flow Integrity) and a safe language or virtualization are all software solution. In next section, the SFI technology is introduced.

3.5 SFI

The Software based Fault Isolation is mainly applied in a single address apace. In the paper *Efficient Software-Based Fault Isolation 1993* , method is described clearly and accurately.

3.5.1 Segment Matching

Segment is a concept introduced in 3.4.2. For the embedded system has no MMU. Segment can also be established by software definition. The program in part of Segment is not allowed to jump out or call to the segment that is separated from it. A direct method is block all the illegal instructions that possibly involved. This method is called SFI. These possible unsafe instructions are a subset of instruction set.
Most control transfer instructions, such as program-counter-relative branches, can be statically verified. Stores to static variables often use an immediate addressing mode and can be statically verified. However, jumps through registers, most commonly used to implement procedure returns, and stores that use a register to hold their target address, can not be statically verified.[Gra93] According to Robert Wahbe's task, the Segment Matching consist of 4 steps. They use a 'dedicated-register' to represent a transition register and a 'scratch register' as target register.

1. **target address → dedicated register**
2. **scratch register → dedicated register + shift offset**
3. **compare segment , dedicated register**

   If not

   Trap

   Or
4. **store from dedicated register (do the instruction)**

This is the process of the segment matching. In step 1, the segment matching is checking the target instruction, and recognize the target address for writing. The target address is put into a dedicated register which is a temporary register. In step 2, the register offset of the addressing modes is added to the dedicated register, which will generate a final address in dedicated register for the target. In step 3, the dedicated register is compared with the segments which allow access. In step 4, the legal instructions will be executed.

3.5.2 Address Sandboxing

To lower the work load of Segment matching method, another technique can be applied. The address sandboxing is not comparing address range but using a segment mask, and generating the method by AND and OR operation.

1. **target regand mask reg → dedicated register**
2. **dedicated register—segment register → dedicated register**

3. store from dedicated register (do the instruction)

Before each unsafe instruction they simply insert code that sets the upper bits of the target address to the correct segment identifier. We call this sandboxing the address. Sandboxing does not catch illegal addresses; it merely prevents them from affecting any fault do main other than the one generating the address. [Gra93] The addressing sandboxing method is for optimizing segment matching. Because simple matching cost so many instructions to do it.

3.5.3 Improvement to SFI

SFI is introduced for a long time. The main thought is still worth referring. The method can be applied in other situation only with a little improvement. The main thought of SFI is to identify the vulnerabilities and insert a function call just before the vulnerable instructions which is based on segment matching or address sandboxing. But the original method only applied to the write operation. For a system without segment technology, a complete software solution is eligible.

As an improvement to SFI, more operation can be inserted into the position of segment matching including more complex access control technology. To apply more cross fault domain operations, the algorithm can also allow some legal operation with some special masks between segments. The function for fault isolation can be extended as an administrator of the the resources of the whole machine.

In the paper of *Efficient Software-Based Fault Isolation 1993* , the optimization is also discussed. The designer reduced the overhead of software encapsulation mechanisms by avoiding arithmetic that computes target addresses directly. Considering the instruction set of the local machine, the fewer cycles are applied, the less overhead of the algorithm.

3.6 DDT

Data Space Descriptor Table (DDT) is a technology for memory isolation instead of the hardware memory separation such as Paging and Segment. It has some advantages.

- Overhead Saving
- Agile and Multifunctional
- Software-Based
- Easy For Administration

The DDT is in a form of continuous data. It restoring the subject info and the memory range for a segment. Comparing to direct matrix or list in resource side or user side, the

DDT organizes the memory spaces in a regimentation. The structure of DDT is array or a chain list that restoring the elements of the Access Matrix in the memory, as Figure 3.3 shown. The memory descriptor is the part identifying the range of a segment. The owner info describes the subject that this segment is in. User info describes the users can use this segment.
The advantage is that looking up and change is much more easier than locating the elements in every function or memory fragment. The administration for the Matrix is easier and also saving overhead comparing to segment separation who are causing severe fragmentation.

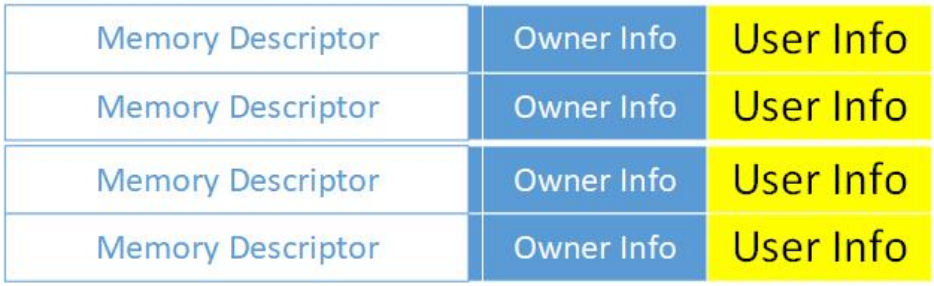

Figure 3.3: Data Descriptor Table

There are many strategies for Memory Descriptor. To define a memory space, two parameters is required. The start address is always necessary. The second address can be end address, size in byte or the next address. These length differ in comparing cost and overlap. The developer shall choose the suitable data structure for different requirement.[Ste16]

3.7 Paravirtualization

Virtualization describes the separation of a resource of request for a service from the underlying physical delivery of that service. [VMW06]
Hardware Virtualization is a technology that a guest system running on the interface looks like a hardware computer. The system that interface build on is called a host. Paravirtualization is a technology applying the thought of virtualization and offering a different interface as the host system. Paravirtualization is a kind of native virtualization, which means the guest system is running native machine code. But some interface in changed and supported by the software. The hypervisor which is the core component of a paravirtualization offers the interface that the guest system required.
It is also valuable for embedded system. The hypervisor has following functions:

- Verifying Used Address in Runtime
- Definition of Virtual Instructions and offer Interface for these Instructions

- Execution of the Virtual Instructions

For embedded systems do not have MPU for hardware memory protection, paravirtualization is a valuable choice. The Figure 3.4 shows a paravirtualization with only hardware and Hypervisor.

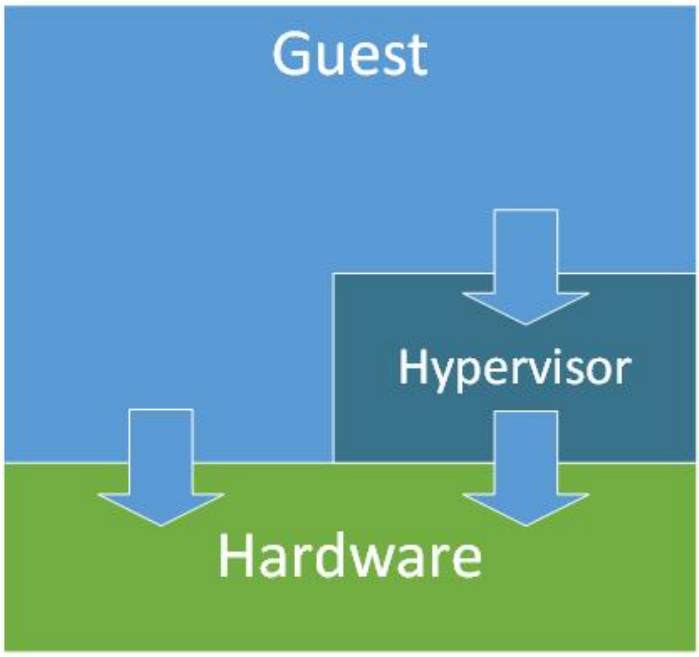

Figure 3.4: A paravirtualization without host operating system

The method to call the interface that the Hypervisor offers is to insert a function call just before every instruction that is virtual, as Figure 3.5 shown.
As shown in Figure 3.4, the Hypervisor is called by function call.

3.7.1 Virtual Instruction

Paravirtualization offers a is similar but not identical to the underlying hardware-software interface.[NM07]. For the embedded system, a paravirtualization is more achievable technology. What is different here is that the calling instruction is not usually a executable instruction, but a virtual instruction instead. They include all the necessary information. As explained in chapter 1, the instructions usually carrying op-code, addressing mode, and registers. These information should be translated into the executable instruction or mapped to the operations, which is called instruction emulation.
The virtual instruction is a subset of the whole instructions. They are organized by the programmer's requirement. To organize a secure system, the first step is to identify a memory separation and to find out these risky instructions which may cause illegal cross

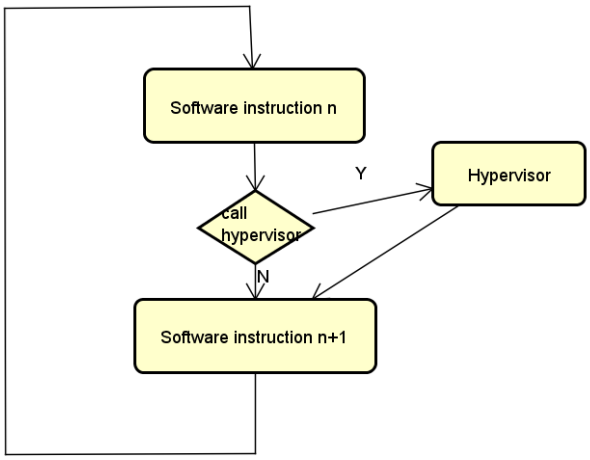

Figure 3.5: Hypervisor Call

module read/write. And replace these instruction by virtual instruction and a hypervisor call before them.

3.7.2 DDT Implement

After getting the information of virtual instruction, the hypervisor verify the address. DDT technology offer a solution. The hypervisor has a centered administration table for memory space. The hypervisor can execute segment verifying as Section 3.5 shown with the Data Descriptor Table. In the multi-module systems, the access authority is clearly recorded in the table and it's also extensible. The table itself and the hypervisor core must be protected. And the memory space of other modules are allocated in the developing time. And these tasks usually done by frameworks or secure compiler.

3.7.3 Runtime Executing and Verifying

The first part of Hypervisor is verifying the address. To do this, the virtual instruction should be processed in certain way. After the verifying success, the instruction will be translate into executable instructions. The last part is executing the translated instruction.
The details of the instruction emulation will be introduced in Chapter 4 and the detail of the Hypervisor implement will be demonstrated in Chapter 5.

4 Instruction Emulation

Instruction Emulation is a technique widely used in modern computers. Instruction Emulation is implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface and functionality.[JS05] For a larger scale system, the developer can choose virtual machine or full set emulation. The embedded system's restriction is the capacity of the system are usually very limited. For embedded system, it is better to lean more on the software-based memory separation and paravirtualization. The embedded system's restriction is the resource of the system are usually very limited. An advantage of full set emulation is that the developer can offer a more different interface for the system. If A language is offering a interface of B system, it is called a A-B emulation, A-B compiling or A-B interpreting.

4.1 JIT vs Interpretor

For instruction emulation, there are two common solutions, Just-In-Time compilation or Interpreting. In this section, the principles of them will be introduced.

4.1.1 Static Compiler

First, let's have a look at the static compiler. Since C language invented, thousands of software is developed with compilers. For a C-Binary compiler, it is in fact a translating from C language text to executable binary code. After compilation, the linker links the library, the program code defined by the compiler. These compiler who translating at one time called static compiler. As Figure 4.1, the compiler only does the first step which translating the higher level language to assembler code. Since assembler can be mapped to binaries. The linker organizes the different parts of the program in a fully executable file.

4.1.2 Just In Time Compilation

Just In Time Compiling, which may also called binary translation at runtime, is a method that is more flexible. The compiler is activated in runtime. When the program start up, the compiler only compiles the functions that is crucial. The binary code is restored in a fixed space which is called code cache. Then every time a function is called, the compiler

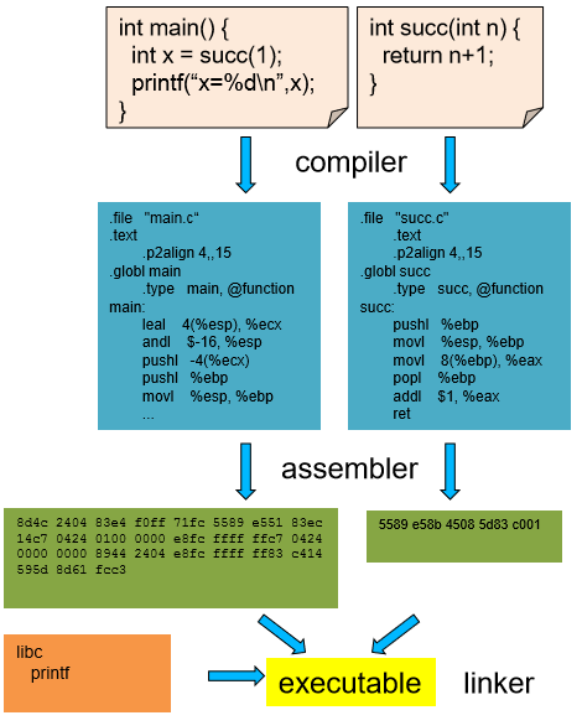

Figure 4.1: Static Compiler [Roh12]

will compile only the called function. After the program is executed, the cache binary code will be deleted.

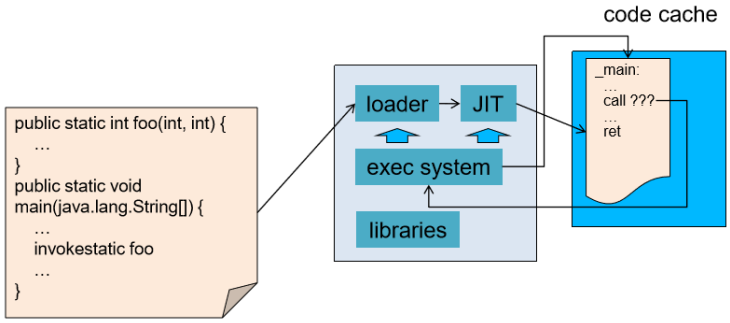

Figure 4.2: Just-in-Time Compiler [Roh12]

4.1.3 Interpreter

Interpreter is more straight forward, the program will not translate any code, but map the code to binary from library directly. The interpreter do not require step before executing,

but it is slower in executing time, as shown in Compiler2. The key difference between JIT and Interpreter is that the compiler will generate text code of the system interface. For example, Javascript and PHP, these languages are interpreting language. They don't generate a executable binary file.

Java, the most wide used OOL (Object Oriented Language), is sometimes also used in embedded system developing. It has great portability for developing. Java as a high level object oriented language, is running on a Java virtual machine (JVM). JVM has a full system abstract including registers and memory. These interfaces are offered by Java binary code. This is the reason why Java is so portable. For the programmers, Java offers always the same language for different platform, but the JVM can be implemented for different platform. Java in fact do two steps. The first is compiling Java language into the Java binary (Java Binary is running on a virtual machine that obeys the specification of JVM). The second is interpreting the Java binary into native machine code. What is worth attention is that Java is not only interpreting Java binary into machine code, but also do Just-In- Time Compiling. It does not always interpret the Java binary. If the function is massively called the JVM will activate the JIT compiler, to generate some native machine code for the calling, which can improve the performance of JVM significantly. Android is also a design whose JVM is running on a Linux kernel.

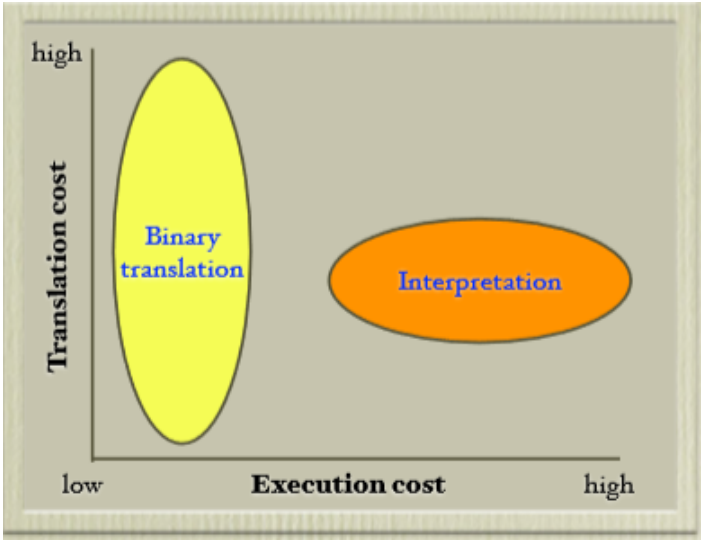

Figure 4.3: JIT vs Interpreter [Sa07]

4.2 Emulation in Virtual Machine

Virtualization can be based on the mechanism of Emulation. The system is running on an virtual machine. The thought is to build abstract system like real machine, and verify the

memory access in emulation. As shown in 4.4, the blue part is virtual part, the green part is machine executable.
If the virtual machine is implemented with an emulation, it looks like a software abstract of a real machine. All the registers are emulated in the memory. The virtual machine load and emulate the instruction from memory and run it just like a real machine. These machine can implement a security mechanism like memory software-based fault isolation. And several virtual machines can be run on the same host concurrently. They have their own resources. Every machine core is logically identical to a real machine, including the virtual registers and memory space. The system is running in the core, and all the programs are written in virtual instruction or real instruction, but analysis is required before execution. In this architecture, the access matrix can be integrated in the system core.
The program code is in blue colour which means they are virtual instructions. The core part of Hypervisor realize two function, verifying the memory access and emulating the instruction. To realize the two functions, many components may be added to the Hypervisor, which depends on the design.

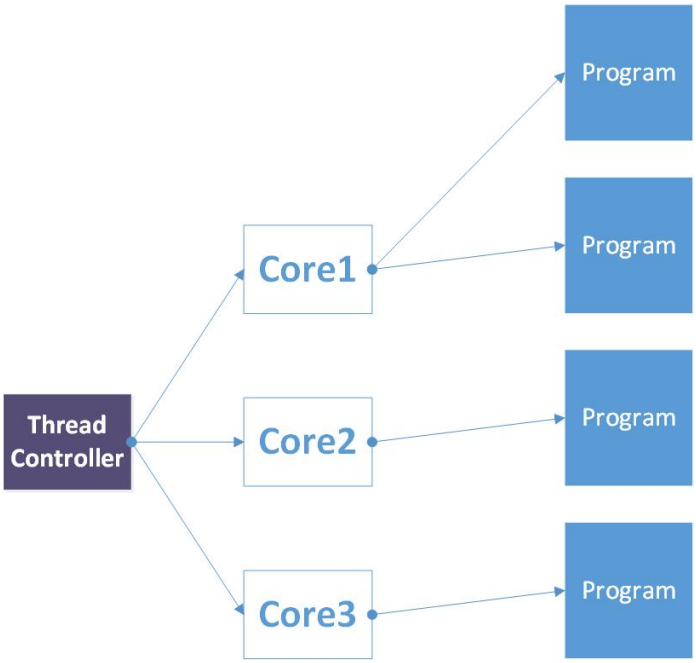

Figure 4.4: Emulation in VM

4.3 Instruction Emulation In Paravirtualization

In this section, the steps of building an instruction emulation for paravirtualization on an embedded system in detail. As informed in Chapter 3, to implement a Paravirtualization on a embedded system, Hypervisor with both functions of verifying and emulation is a key

component. Considering the complexity of the instruction set and addressing mode, only verifying is not enough, or in another word, not suitable for development. For example, when both of the source and destination of an instruction are pointers or indexes, we need to verify both of the them, which is unwise for development.
In an executable machine code program, there are some virtual instructions no matter generated by the firmware compiler. All the system generated instructions require this procedure to realize cross domain action. As Figure 4.5 shown, the light part is a virtual instruction and the Hypervisor is called.

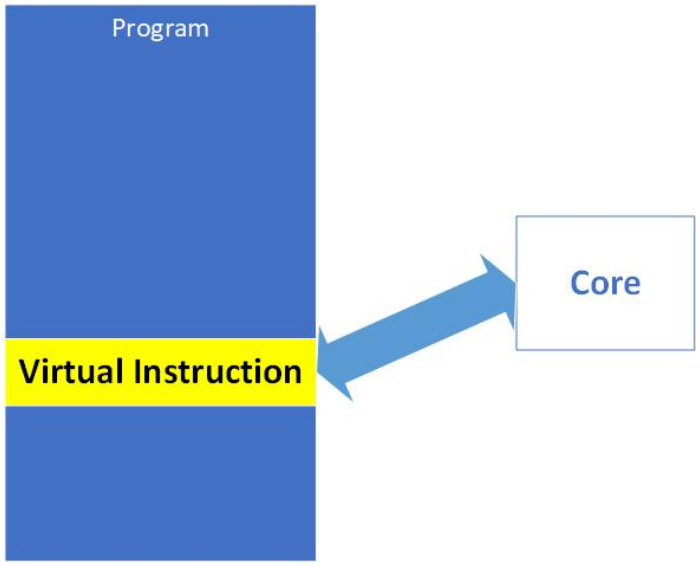

Figure 4.5: Hypervisor of Paravirtualization

4.3.1 Instruction Set Design

If the system requires a full-set emulation, the whole instruction set should be chosen for the emulation. If only a paravirtualization is required. It is important is to choose the subset of the instruction set. As informed in Chapter 3, the paravirtualization require a subset of vulnerable instructions of the whole set. For the Hypervisor, the vulnerable instruction is some cross-domain instruction that can be detected only at runtime. It is hard to define by static methods like compilers or frameworks. For the reason of memory separation of firmware, the subset of cross domain read/write/call instruction should be virtualized and remap into instruction steps that is more convenient for management.
Here is a design for memory separation mechanism. Vulnerable operation is the instructions using an index or pointer. To verify these instructions, it is impossible in static compiler. This leads us to build an instruction-instruction emulation. And the procedure of verification can be integrated with emulation. The reason not to choose a full virtualization

on a embedded system is that the virtualization require lots of resources. A subset of full instruction set in required only. The minimum set of vulnerable subset is:

- Double Operand Instruction with pointer or index
- POP/PUSH offset(register)
- CALL offset(register)
- RET offset(register)

It's some possibility to solve the problem that nearly full set is splitting. The thought is to replace those cross-domain instructions with basic instructions. In the process of emulation, R4 is used as the temporary register, and not be referred by programs. As different addressing mode are existing, this double operand instructions can be optimized as:

1. **MOV offset(S) offset(D)** ⇒ ***vload offset(S)***
   vstore offset(d)

2. **MOV Src offset(D)** ⇒ **MOV Src R4**
   vstore offset(D)

3. **SUB offset(S) offset(D)** ⇒ ***vload offset(S)***
   XOR offset(S) FFFFh
   ADD offset(D) R4
   vstore offset(D)

The original instruction on the left side of the arrow can be replaced by the instructions of the right side. Not the whole set is shown here. As the conclusion, the vulnerable instructions can be replaced with a virtual instruction set.

- vload src
- vstore dst
- vpush src
- vpop dst
- vcall dst

4.3.2 Interpreting Or Binary Translation

As discussed in Section 4.1. There are two method which can be used for runtime emulation, the Interpreting or Binary Translation (JIT). They have both advantages and disadvantage. A Just-In-time compiler require more steps to emulate. Interpreting is straight forward but low efficiency in long-term running. The detail will be discussed in Chapter 5.

4.3.3 Procedure of Instruction Emulation

This section is detailed step that an emulation for paravirtualization works.

- Allocate the Space for Binary

  The memory space of involved instructions should be allocated first. The aim is to allocate the space for virtual instruction and to allocate the space for translated instructions or interpreted instructions. As informed in Section 4.1.2, an A-B JIT will generate some executable code for the B system. These code must be located in a space for which the Hypervisor has a convention. In a Von-Neumann architecture system, the instruction is shared spaces with data, so that we can define a fixed or variable length array. The array entrance can be fixed or dynamic. And when the array address is called, the system will execute the content of the array in order. The space of instructions should be managed dynamically to ensure the system running smoothly.

- Get the Virtual Instruction

  When the program is emulating a instruction, the address of virtual instruction that is to be emulated should be picked from a fixed. No matter the emulation or paravirtualization is implemented, the registers would inform us current virtual instruction in different way. As an example, the Paravirtualization is using a function call, so that the instruction would be indicated by the stack. In the other side, a full emulation will get a virtual program counter to the address of the object instruction.

- Analyze the Virtual Instruction

  After getting the instruction, operation are executed to the instruction based on the definition of the instructions.

- Apply Operation for the Instruction

  The interpreter or the Just-in-time compiler do different operation in this step, the JIT needs a binary translation into the fixed space. Interpreter only calls the function that is to be executed. After the operation executed, the allocated space should be re-allocated.

5 Implement Details

In this chapter, the real implementation of Hypervisor will be discussed. The flow chart of the implement is given in the appendix on Page 44.
As discussed in Chapter 3, there are some benefits for using a Paravirtualization. The firmware is running with machine code in most of time. Hypervisor is only called when cross domain instructions are used.
As shown in Figure 5.1, this is the procedure in detail. A call instruction is used to ask the Hypervisor. The registers must be saved into a cache, because the register will change as the Hypervisor is running. As introduced in Chapter 2 and 3, the function call will restore the address of the virtual instruction in the stack, so that the second task of the entrance is to get the virtual instruction into the Hypervisor cache space. These cache spaces, an array or list, shall be allocated by programmer. Then as the SFI does, an address verifying is done. it have to be analyzed for the obtained virtual instruction and evaluated whether the operand address is in the legal range. Then the virtual instruction is emulated by binary translation or interpreter.

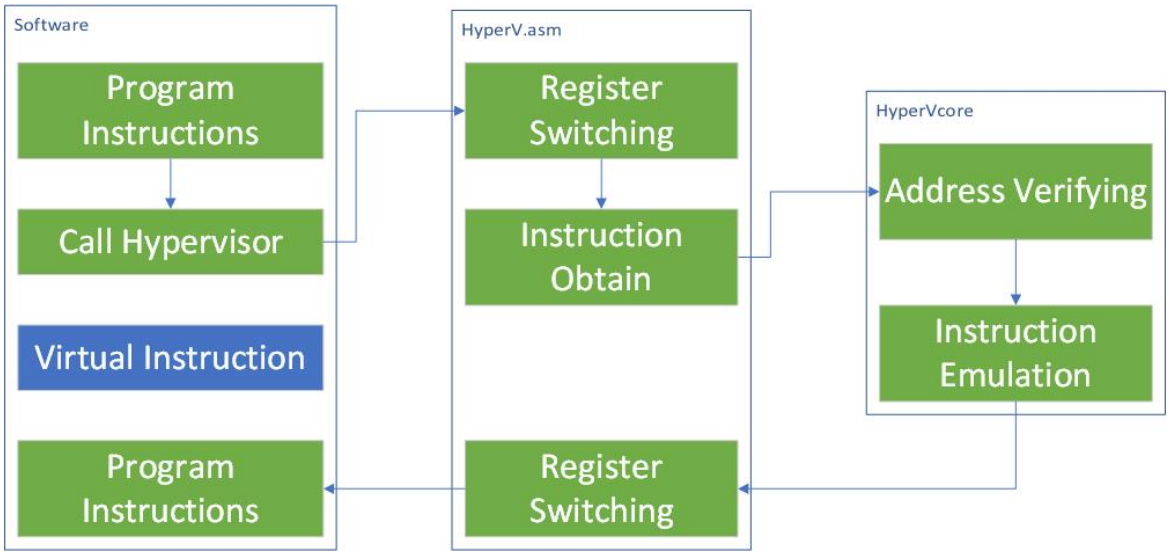

Figure 5.1: Acticity of Hypervisor

5.1 Register Switching

As Figure 5.2 shown, the registers are restored into an allocated memory space. The register switching is necessary because the Hypervisor will do the operation which would change the original status of the machine. The safest method is to save all the general purpose registers, and do emulation operation among these saved registers. When the program leaves Hypervisor, a reverse step will be done, putting the saved registers back to the CPU. This step can hardly be written in higher-level language. The program can be written directly in assembler.

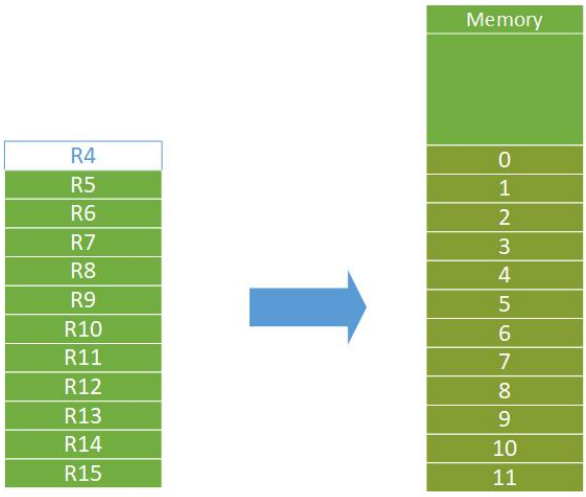

Figure 5.2: Switching Register

The dedicated register is introduced in SFI in section 3.5. In Figure 5.2, R4 is a dedicated register. It could be flexible that the dedicated register is a real CPU register or a memory cache, which means that the virtual instruction can be emulated by using a CPU register also. The other general purpose registers used in the program must be restored and kept original.

5.2 Instruction Obtaining

As shown in Figure 5.3 , the Saved return address is a pointer of the next instruction after a function call. The return address is used as a return position of a function call, but here the return address is different from a normal function call. The return position should be an instruction after the return address. The true return address should be calculated by the emulation, determined by different lengths of the virtual instructions. The virtual instruction obtained should be put into a variable and be analyzed in emulation.

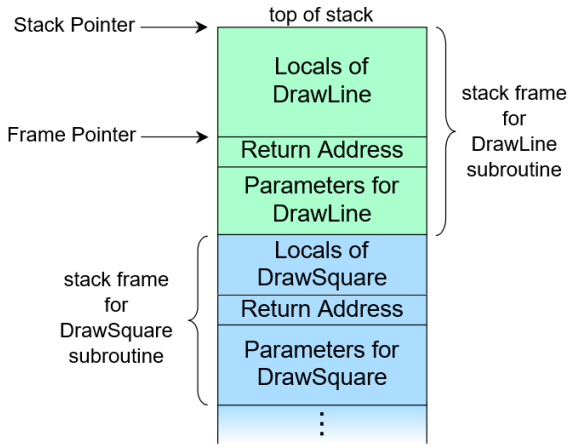

Figure 5.3: The stack of a function call [Sha17]

5.3 Instruction Emulation

As introduced in Section 3.7, paravirtualization offers an interface for the software layer. In Section 4.3, the set of virtual instructions is designed for the cross domain operations. The selection of the set is based on Software-Based Fault Isolation. Now it is implemented in MSP430.

5.3.1 Virtual Instruction Set

The instructions in this set have their own Op-code and addressing mode which can be analyzed by the Hypervisor. Here is an example on MSP430 for this set: *vload* is doing

SYNTAX	vload									B/W	Ad mode		S reg		
BITS	0	1	0	0	0	0	1	0	0						
SYNTAX	vstore									B/W	Ad mode				
BITS	0	1	0	0	0	0	1	1	0						
SYNTAX	vpush									B/W	Ad mode		S reg		
BITS	0	0	0	1	0	0	1	0	0						
SYNTAX	vpop												D reg		
BITS	0	1	0	0	0	0	0	1	1		1	1			
SYNTAX	vcall									B/W	Ad mode		D reg		
BITS	0	0	0	1	0	0	1	0	1						

Figure 5.4: Virtual Instruction Map & Structure

an action that moves data into R4 after verifying. *vstore* is doing the reverse that move data from R4 after verifying. They presenting the access of writing and reading. *vpop* and *vpush* is presenting the writing /reading access to the stack. *vcall* is preventing cross domain function call which is out-of-authorized. For different addressing mode, the length of virtual

instruction may be 16 bits or 32 bits.
As the 5.4 shown, the instructions mainly keep the structure of machine code. *vcall* and *vpush* are executable. But they also require emulation if we do the operation in the register cache but not in CPU. This will be shown in 5.5.

5.3.2 Emulation with Interpreter

As informed in Chapter 4, the emulation methods are interpretation and binary translation. In Figure 5.5, the interpreter is a part of the Hypervisor, after the instruction analysis, the Hypervisor calls the interpreting function directly. The operation of the virtual instruction is executed with the cached register introduced in Section 5.1 and the address of the operand, which is all in the memory. As shown in 5.5, the system core integrated with interpreter is

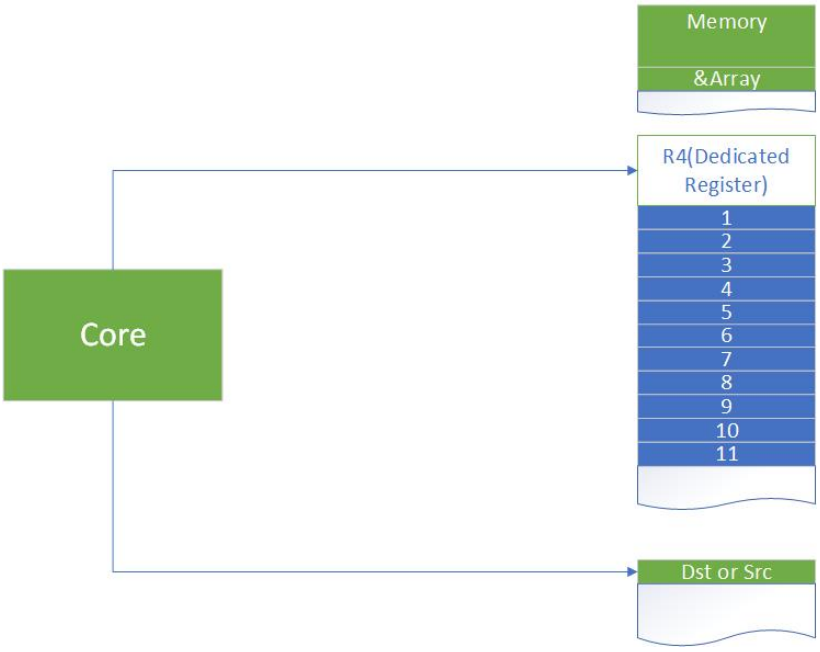

Figure 5.5: Emulation with Interpreter

doing the manipulations among the register cache.

5.3.3 Emulation with Binary Translation

A binary translation would be different. The Hypervisor core generates machine code for emulation after the analysis of the virtual instruction, which means additional code is doing the execution of the virtual instruction. Options still exists that the operation is done in the register cache or CPU. An example that is doing operation between memory and CPU is shown in Figure 5.6. But the data in R4 must remain safe and unchanged until the Hypervisor call is finished.

The difference between binary translation and interpreter is illustrated in chapter 4. Here in the Hypervisor design, binary translation still do more step in generating code for emulation. And the interpreter require more step in execution. Overall, here an instruction-instruction emulation is working, so that the binary translation has not so much advantage. If binary is kept and reused for next execution or the instruction emulation is implemented with highly encapsulated instructions, binary translation will have its advantage.

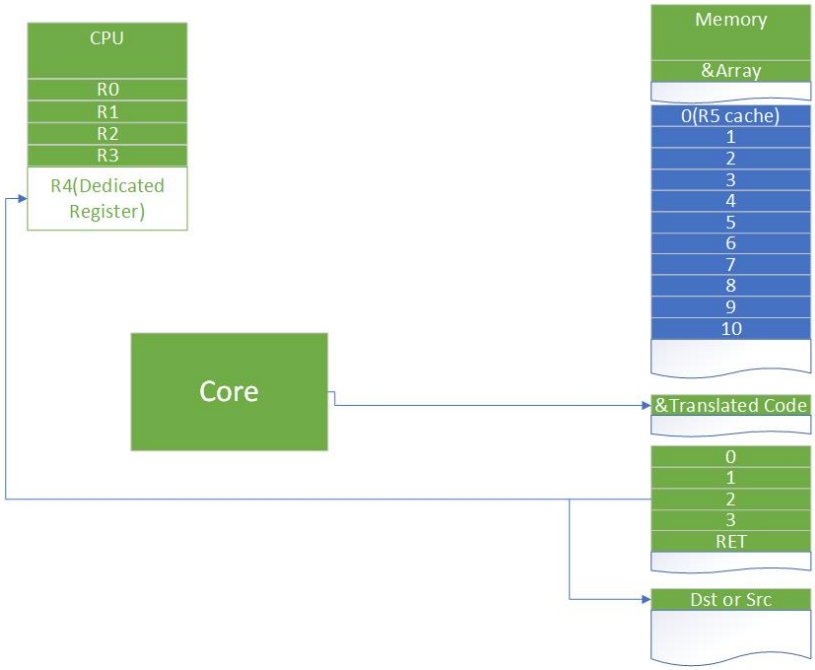

Figure 5.6: Emulation with Binary Translation

As shown in Figure 5.6, the system core only generates some binaries, and the manipulations are executed by executing the binaries.

5.4 Address Verification

This section introduces the technique of the Address Verification in Hypervisor. General concepts of address verification were introduced in Section 3.5. According to Section 3.5, Segment Matching is a straight method. When the virtual instruction is using a indexed mode, the first step is move the target address into the dedicated register. Then the dedicated register should be added with the shift offset of index mode. The dedicated is now carrying the target address. This step is called pre-verification. Then the dedicate register should be sent into a verifying function as a parameter which will tell if the sent address obeys the Data Space Descriptor Table.

6 Evaluation

In this chapter, the environment to develop and test the Hypervisor on MSP430 and the evaluation about the implementation are introduced. The environment enables developer to develop with PC platform and test in a relatively simple way. The chapter 3 and Chapter 4 tells the initiative of the implementation. The evaluation tells the practicability of our design.

6.1 Environment

First, software that is required for the MSP430 development is introduced.

6.1.1 Embedded System Development

Software development on embedded system has several procedures. If you are using a higher-level language but not binary or assembler. Compilation or interpretation of the programming language is quite necessary. To develop an embedded system, the first is the firmware. As introduced in Chapter 4, dynamic emulation needs interpreter or translation scheme. The most basic method is a static compiler. The C compiler for the embedded system will first compile C files into Object files. Then the linker will link them together. At last the locator will place the binary into the memory.

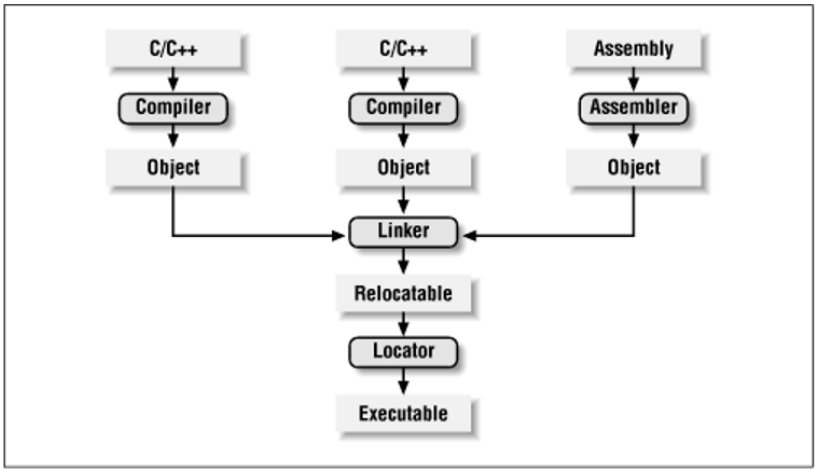

Figure 6.1: Embedded Development Procedure [Bar99]

The compiler translates the language into binaries. The linker combines the binaries that generated by the compiler. The locater determines the address in the memory for the firmware. The three steps create and upload the firmware for the embedded system.

6.1.2 GCC

GCC (GNU Compiler Collection), is a comprehensive compiler by GNU project, which enables a lot of programming language. It also supports a lot of system including ARM, x86. The GCC enables even Java-Binary compiler. C language is widely used in embedded system development. GCC supports the syntax of standard C99. Some features is important for the development.

One of the important point on this software is in-line assembler in C programming. These Assembly codes are compiled as macro (text command) and they will map into machine code later. asm() is a key word in C language. These assembler in C file will be compiler in a different way as they look like. The assembler syntax is integrated the C program, so that the assembler code will be translated into executable instructions. These translations is ambiguous in different situation, which also limited the operation you can do in the in-line assembler. Nevertheless, in-line assembler is still a useful tool for development.

The GCC compiler can also generate assembler file whose suffix is .asm. The code written in .asm file will be mapped directly into binary code as the platform feature.

TI company also offers a compiler and locater,TI v16.9 can be found in the Code composer. The latest GNU v6.2 and TI v16.9 are both available for the development on MSP430.

Pre-process function gives GCC a comfortable developer support that finds the evident mistakes in syntax.

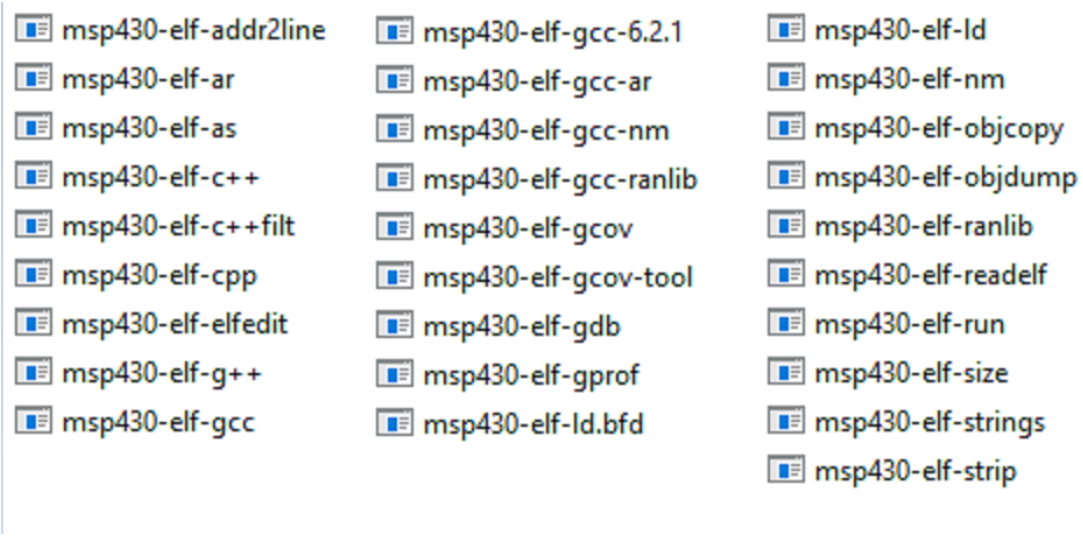

Figure 6.2: Compiler Components

On windows platform, developer can run compiler directly by the executable file as Figure 6.2 shown. These are the tools including different compilers, deassembler and others. Different Compiling option can be apply in command line. To compile a C file into assembler, apply *msp430-elf-gcc-6.2.1.exe -s main.c*
In Figure 6.3, an example of using a objdump to analyze a linked or located binary file is given.

```
PS C:\Users\tg> cd C:\ti\ccsv7\tools\compiler\msp430-gcc-6.2.1.16_win32\bin
PS C:\ti\ccsv7\tools\compiler\msp430-gcc-6.2.1.16_win32\bin> .\msp430-elf-objdump.exe -d C:\Users\tg\workspace_v7\Hypervisor\Debug\main.o

C:\Users\tg\workspace_v7\Hypervisor\Debug\main.o:     file format elf32-msp430


Disassembly of section .text:

00000000 <main>:
   0:   b1 00 02 00     suba    #2,     r1      ;

00000004 <.LCFI0>:
   4:   81 4c 00 00     mov     r12,    0(r1)   ;

00000008 <.Loc.11.1>:
   8:   40 18 b2 40     movx.w  #23168, &0x00000;0x05a80
   c:   80 5a 00 00

00000010 <.Loc.13.1>:
  10:   b0 13 00 00     calla   #0              ;0x00000

00000014 <.Loc.14.1>:
  14:   15 43           mov     #1,     r5      ;r3 As==01
  16:   02 00           mova    @r0,    r2      ;
  18:   4c 43           clr.b   r12             ;

0000001a <.Loc.36.1>:
  1a:   a1 00 02 00     adda    #2,     r1      ;
  1e:   10 01           reta                    ;
PS C:\ti\ccsv7\tools\compiler\msp430-gcc-6.2.1.16_win32\bin>
```

Figure 6.3: A Disassembly sample

The command: *msp430-elf-objdump.exe -d main.o*
To link a executable file, GNU make is required. Gmake (GNU make) processes directives of a makefile to build a binary. On windows platform, to make your file, a make file have to be created with the make file pattern. Then run the command to create the executable file. GNU make can generate the executable file, after make file is organized in the project folder.
For example, *gmake -k -j 4 all -O*

6.1.3 Code Composer Studio

Code Composer Studio (CCS) can be downloaded from the website of TI. CCS supports a lot of embedded system produced by TI, including MSP430. It is a modern and comfortable IDE for the development on these micro-controller including MSP430. Code Composer integrates TI compiler, GNU compiler, project management, driver for MSP430 development board and user interface for debugging. Code Composer Studio combines the advantages of the Eclipse software framework with advanced embedded debug capabilities from TI resulting in a compelling feature-rich development environment for embedded development. [CCS17]

Developing with real board and Code Composer Studio, would be much more easier than compiling and linking manually with Gmake and compilers.

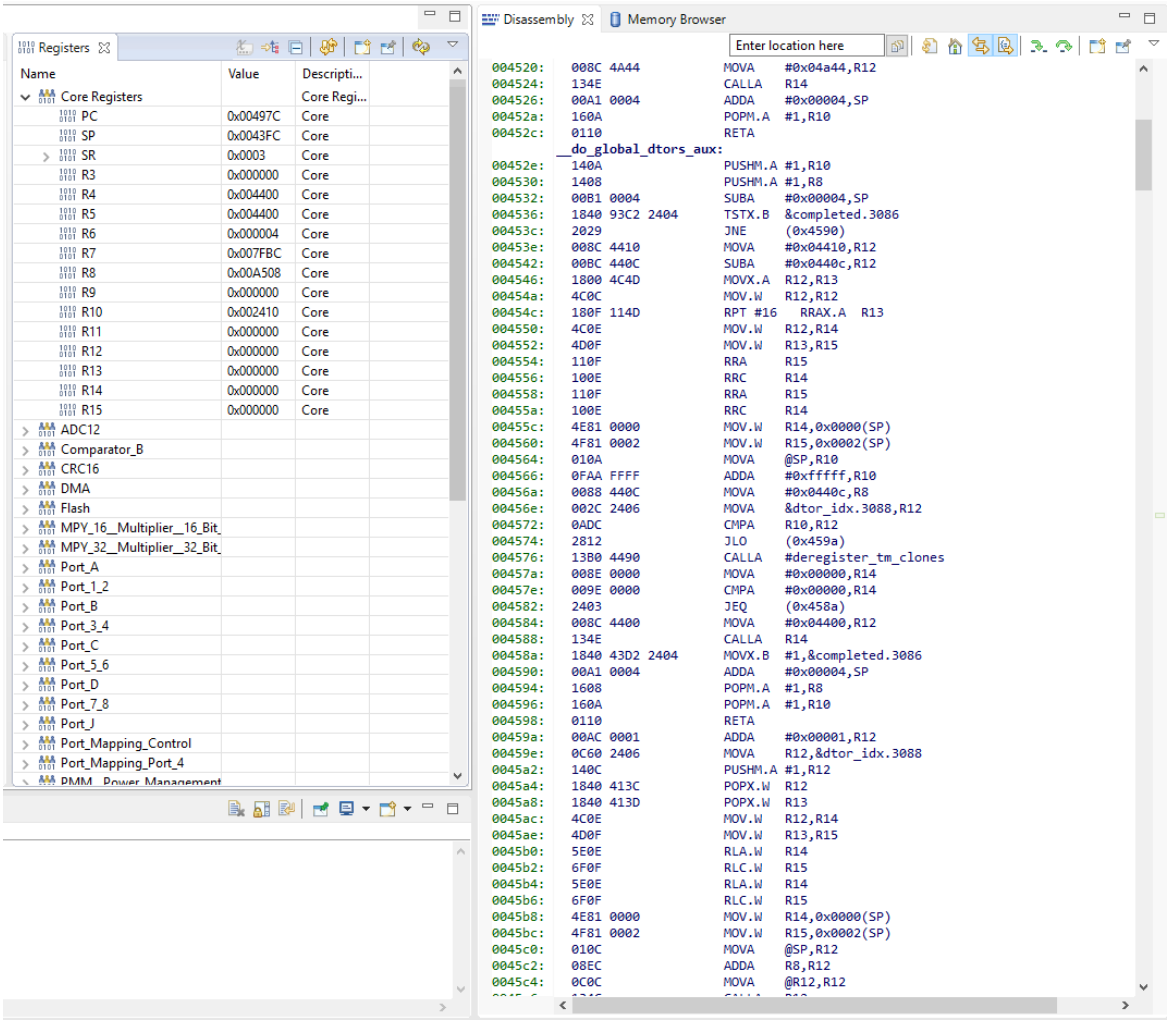

Figure 6.4: A Code Composor Studio Tools

As shown in 6.4, CCS can burn-in to launch pad. The drivers for the launchpad is integrated in CCS. The firmware development debugs on disassembled binary file, which is located in the right window. CCS provides a fully visualized user interface for development. CCS can also monitor the registers in the left window step by step.

6.1.4 MSPsim

Another software is the MSPsim, which is a simple tool simulating MSP430 systems. It is a virtual machine written in Java that offers the interface of MSP430 platform. MSPSim gives several example firmwares including Tmote Sky sensor mode which is an embedded system. MSPSim supports the development on Windows PC. After the compilation, linking

and addressing process, the machine code for the firmware will be generated which could be installed by copying directly into the directory. The source file of MSPSim can be found on Github. To install the MSPSim, GNU make and Java is required. After the files of MSPsim prepared, in the target folder, enter

make

to install the MSPSim.

Then, you can start different mode for the simulator. For example,

make runsky

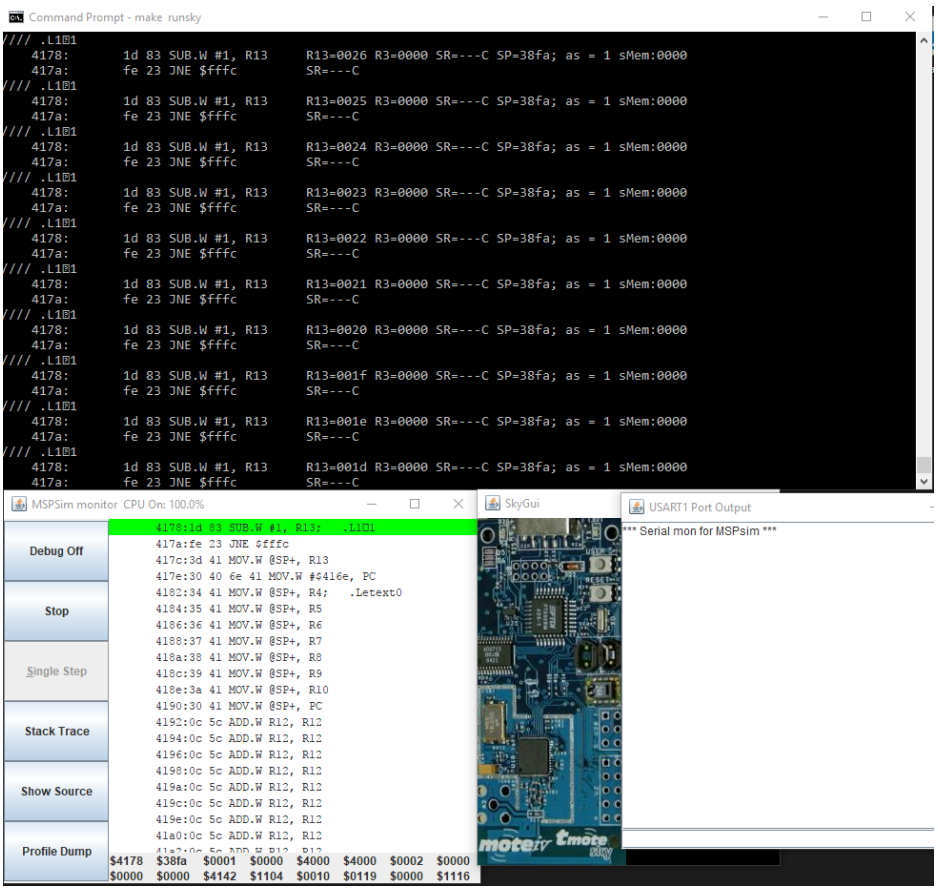

Figure 6.5: MSPSim Debugging

The firmware is installed in the folder /firmware/sky/ . Developer can replace the firmware with their own.

6.2 Result

By using these tools introduced in this Chapter, the firmware can be analyzed by each instruction. First, the cost of executing the virtual instruction is counted. Second, an average

appearance of the virtual instruction is counted. In this section, a Hypervisor implemented binary translation is evaluated.

6.2.1 Instruction Cost

The cost of an instruction emulation is the instruction count that is executed to emulate the virtual instruction . In table 6.1, the costs by components for different instructions are listed. In this table, the Hypervisor parts introduced in chapter 5 are involved in the counting (Measurement). The part of obtaining virtual instruction and op-code analysis are Pre-analysis. The address obtaining and call verification are Pre-verification.

Virtual Instruction	Register Switching	Pre-analysis	Translation	Pre-verification	Emulation	**Total**
vload(index)	16+15	26	16+10	15	5	103
vload(indirect)	16+15	26	14+6	11	5	93
vload(autoincrement)	16+15	26	14+6	11	5	93
vstore(index)	16+15	26	14+11	13	5	100
vpop(index)	16+15	26	14+10	13	5	99
vpush(index)	16+15	26	10+11	13	4	95
vcall(index)	16+15	26	10+11	13	4	95

Table 6.1: Cost of Instruction Emulation

The instruction cost is close to 100 instructions, which means it is close to 100 times of original cost. If the address verifying mechanism is added, the cost would be larger. The list shows that the inevitable costs are more than half. Only Translation and emulation cost can be reduced.

6.2.2 Virtual Instruction Rate

In this section, several firmwares are tested to know the rate of vulnerable instructions. As introduced in Chapter 4, the vulnerable instruction are these instructions who could cross domain and can not be checked statically at compile time. All the registers using indexed mode and indirect mode(including autoincrement mode) are the vulnerable and would cause a Hypervisor call. In sum, 3 firmwares and 7882 instructions are included in the test. The data is shown in 6.2.

Firmware	Sample	Indexed mode	Indirect mode	**Percentage**
Tmote sky	2885	209	65	
Tmote esb	2423	245	98	
blink.z1	2574	293	59	
Total	7882	747	222	12%

Table 6.2: Rate of Vulnerable Instruction

In the whole 7882 instructions, there are 747 instructions in indexed mode and 222 instructions in indirect mode. This would cause 969 Hypervisor calls, which means that the 7882 instructions have about 12% virtual instructions. (Here the instructions are assumed to have no indexed source and indexed destination at the same time. The rate is hardly influenced.)

6.3 Estimation

In this section, three estimations will be made. The first is comparing the performance of normal system and system implementing paravirtualization. The second is estimating whether the emulating method is important for paravirtualization. The third is estimating whether the emulation or paravirtualization is better in performance.

6.3.1 Performance of Paravirtualization

To estimate the performance, the result in Section 6.2 is used. An equation is introduced here. Here, the performance of a machine is average time to executed an instruction. Let P_h be the overall performance of a system with paravirtualization. Let C be the average cost of virtual instruction. Let P_m be the performance of a bare machine. Let r be the rate of the virtual instruction. From weighted average,

$$P_h = P_m C r + P_m(1 - r) \qquad , 0 < r < 1 \tag{6.1}$$

From Equation 6.1, it can be got, when $C \gg 1$,

$$P_h / P_m \approx Cr \tag{6.2}$$

From Equation 6.2, we can get that when the cost is dozens of instructions, the execution speed is Cr times slower. If the data in Section 6.2 is used, the rate of work would be about 10 times as slow as a bare machine with no security kernel. If we apply a full set emulation like Section 4.2 introduced. The equation would be

$$P_h / P_m = C \tag{6.3}$$

In this case, if we can optimize the virtual machine which can make the Cost lower than 10 instructions, it would be better than the paravirtualization.

6.3.2 Performance of Instruction Emulation

In this section, the data of the implement shown in Section 6.2.1 is used. As shown in 6.6, more than half of the instruction in the Hypervisor is inevitable unless the structure or

virtual instruction set is changed. The parts that can be optimized is the translation and emulation. They are the left part of the chart.

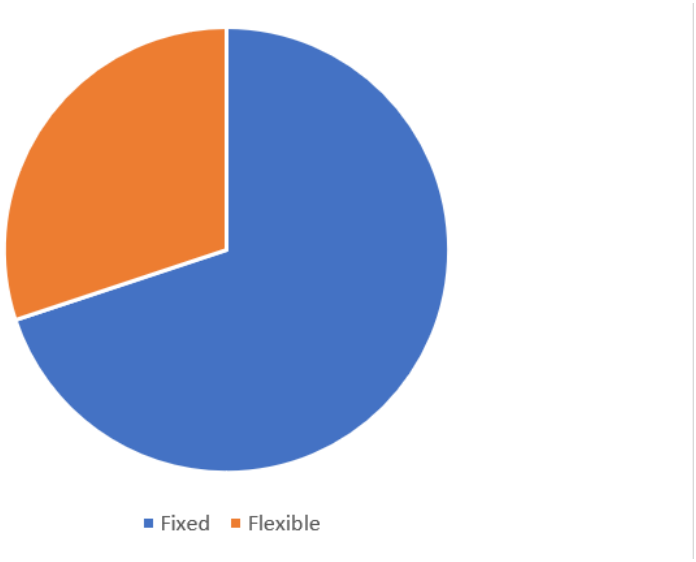

Figure 6.6: The Rate of Fixed Instruction in Paravirtualization

The further conclusion is that if we apply an interpreter in this tiny instruction emulation. The efficiency would be barely improved.

7 Conclusions and outlook

In this thesis, the procedure of designing and implementing an instruction emulation is presented. The instruction set of MSP430 was introduced. In Chapter 3, the reason and method to establish a security mechanism in embedded system was introduced. The core methods is access control and memory separation. Based on these two methods, the Software-Based Fault Isolation was introduced. In the modern virtualization technology, paravirtualization is a suitable framework to implement Software-Based Fault Isolation. Comparing to emulation, paravirtualization is a native virtualization with its advantage that the performance is better when the virtual instructions are few. The emulation technique is introduced in Chapter 4, interpreter and Just-in-Time Compiler can be used in emulation. Instruction emulation also required in paravirtualization in this thesis. The implement detail was given in Chapter 5, this implement is a example of security-enhanced embedded system by pavavirtualization and Software-Based Fault Isolation. In Chapter 6, the develop and test environment is given. Then the evaluation of the implement was carried out. The estimation gives us a view of the efficiency.

Looking into the future, the Instruction Emulation can be optimized in three aspect. First, the algorithm of address verification can be implemented in an efficient way. Second, emulation method can be improved or simplified. Third, the method to implement a security enhanced embedded system can be improve. A full set emulation or other virtualization technology can be applied. A more complex operating system can also be implement through this framework. Modern embedded systems have multi-task schemes. An efficient security mechanism should be put forward.

8 Acknowledgements

This section contains acknowledgements from the author of this document.
I want to thank Prof. Dr. Stecklina. As my supervisor, he did a lot of effort to let me get deep into the topic. With many discussions and corrections, my understanding in this topic is much more deeper. Even for the major of information technology, my knowledge improved for a large step. He is easy of approach. He is not only a supervisor but also a friend. His knowledge helps me a lot for this thesis.
I want to thank Mr. Zhaowen Gong. As my friend, when I was writing this thesis, he was studying in Stuttgart University. He discussed the topic with me, and also had looking for mistakes in my thesis. With his help, I did better in the period of writing the thesis.
I want to thank Mr. Tianfei Gu. He was studying in University of Shanghai for Science and Technology. His support also helped me a lot in the period of writing this thesis. He spent time discussing scientific topic with me, also, these gave me many useful thinking for the thesis.

Appendix: Flow Chart of the Hypervisor Implement

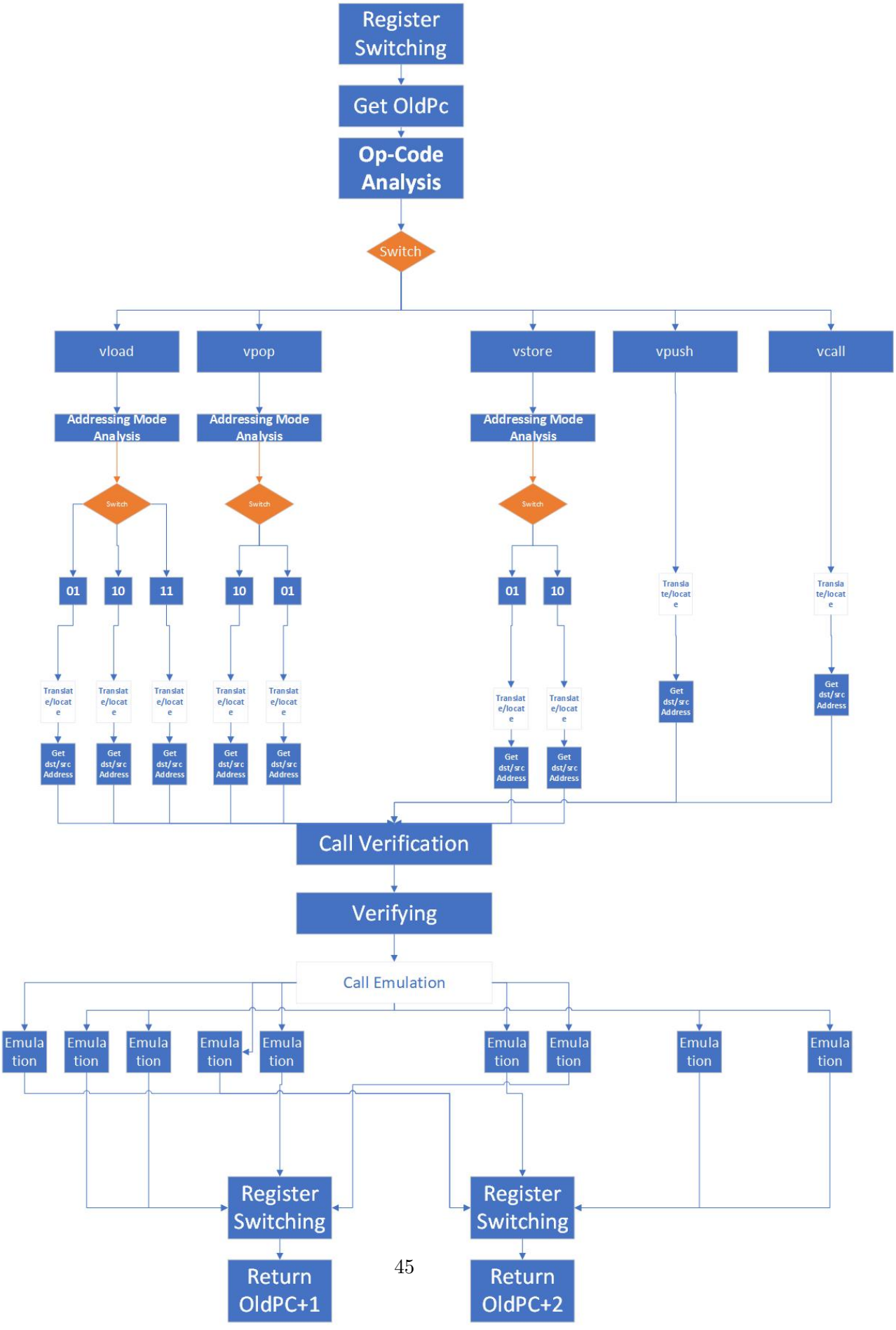

Figure 1: Flowchart of Hypervisor with RPFC

List of Figures

List of Tables

Acronyms

TI	Texas Instruments
CPU	Centre Processing Unit
RAM	Random-Access Memory
ROM	Read-Only Memory
RISC	Reduced Instruction Set Computer
ACLK	Auxiliary Clock
ACLK	Sub-System Master Clock
MCLK	Master Clock
PC	Program Counter
SP	Stack Pointer
SR	Status Register
WSN	Wireless Sensor Network
RBAC	Role-Based Access Control
MAC	Mandatory Access Control
MMU	Memory Management Unit
MPU	Memory Protection Unit
SFI	Software-Based Fault Isolation
VIS	Virtual Instruction Set
CFI	Control Flow Integrity
DDT	Data Space Descriptor Table
JIT	Just-in-Time Compiler
VM	Virtual Machine
JVM	Java Virtual Machine
OOL	Object Oriented Programming Language
GCC	GNU Compiler Collection
Gmake	GNU make file
CCS	Code Composer Studio

Bibliography

[Bar99] Barr, Michael: *Programming Embedded Systems in C and C++*. O'Reilly, Sebastopol, 1999.

[Ben11] Bendersky, Eli: *Where the top of the stack is on x86*, 2011. https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86.

[CCS17] CCSTUDIO: *Code Composer Studio (CCS) Integrated Development Environment (IDE)*, 2017. http://www.ti.com/tool/CCSTUDIO#descriptionArea.

[Gas88] Gasser, Morrie: *Bulding a Secure Computer System*. Van Nostrand Reinhold, New York, 1988.

[Gra93] Graham, Robert Wahbe Steven LuccoThomas E. AndersonSusan L.: *Efficient Software-Based Fault Isolation*. 1993.

[Inc10] Inc., Apple: *IA-32 Function Calling Conventions*, 2010. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/LowLevelABI/130-IA-32_Function_Calling_Conventions/IA32.html.

[Ins13] Instruments, Texas: *MSP430x2xx Family User's Guide*, 2013. http://www.ti.com/lit/ug/slau144j/slau144j.pdf.

[JS05] Jim Smith, Ravi Nair: *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier Inc., San Francisco, 2005.

[NM07] Nakajima, Jun and Asit K. Mallick: *Hybrid-VirtualizationâEnhanced Virtualization for Linux*. 2007.

[Roh12] Rohou, Erven: *Introduction to Compilation, Just-in-Time Compilers and Virtual Machines*, 2012. http://www.enseignement.polytechnique.fr/informatique/INF422/INF422_Compilation.pdf.

[Sa07] Srisa-an, Witawas: *Emulation: Interpretation*, 2007. http://cse.unl.edu/~witty/class/embedded/material/note/emulation.pdf.

[Sha17] Shaw, R. S.: *Call stack*, 2017. https://en.wikipedia.org/wiki/Call_stack.

[Ste16] Stecklina, Oliver: *A Secure Isolation of Software Activities in Tiny Scale Systems*. 2016.

[VMW06] VMWare: *Virtualization Overview*, 2006. `https://www.vmware.com/pdf/virtualization.pdf`.