Hardware Security - Summary

An Open-Source Summary

Thomas Debelle

KU LEUVEN

September 25, 2025

Contents

1   Trust what is it ?

Trust (R. Anderson in "Security Engineering", after NSA): * "Trusted system or component is one whose failure can break the security policy, while a trustworthy system or component is one that won't fail."

Trust (Trusted Computing Group): * "An entity can be trusted if it always behaves in the expected manner for the intended purpose."

So if we trust a component, usually, if it is compromised or something goes wrong inside, then the rest of the trust chain is compromised. So reduce the amount of trusted component is key.

1.1   Evolution of the trust model

Before, internet was much simpler qnd we could assume that the communication link was the untrusted environment and each device was trustworthy. Each was protected by strong cryptographic encryption making them seen as black boxes.

Now, every device can be at risk. Each device is part of the big triangle tradeoff between flexibility-energy-performance. We don't only need secure algorithm but secure *implementation*.

But typically, an implementation can leak a secret as they don't perform the same operation each time which could give the attacker a hint about what it is currently doing.

So we need to be **lightweight and resistant**. Harsh constraint.

1.2 Design methods

We need to apply security at all level ! A **root of trust** is a component at lower level that ensures that anything build on top will be secure.

We try to reduce this footprint of attack and model what the attacker can or could do on your root of trust. Then we need to build some interface to communicate with higher level.

2 Hardware attacks

2.1 Overview

As seen before, now everything is not secure or cannot be trusted. A system is as secure as its weakest link and any attacker will reach out first for the easiest link.

There is a wide range of what an attacker can do with physical access to the circuit, from non-invasive to invasive or from passive to active.

2.1.1 Side channel attack

Passive we simply observe what is going on. So even secure algorithm can be broken if they don't behave like a black box.

We can either check the execution time, the instantaneous power, the Electromagnetic emanations, sound, temperature, …

There exists also some more invasive side channel attack like photon detection (we need to scratch the package to reveal a transistor and see if it emits) or directly invasive such as micro-probing.

2.1.1.1 Timing attack   If we can divide a hard problem into multiple simple one then we win. typically we can check the timing attack on a digipad for example.

So a lot of cryptographic algorithm that leverage from repeating simple step but assuming only the complicated result will be found is no longer valid.

2.1.2 Fault attack

We *actively* try to make the software or hardware bug to skip or reveal secrets. This is part of some *Single Events Upsets* where we could have bitflips.

Again we can have some non-invasive such as *clock glitches, voltages spikes, underpowering,…*

To more invasive such as optical or EM fault injection where we need to scratch the package and/or the shielding to make it happen.

Typically we can perform some **Differential Fault Analysis** where we have the input output result and then we try again with a fault. Sometimes it can be enough to find the key !

2.1.3 Circuit Modification

It is far more complicated and we bypass some traces, . . .

2.2 Power Analysis Attack

We use the **instantaneous** power over time to conduct our attack. We can derivate some information based on the power consumption since larger power consumption can indicate some transition. We can use some simple shunt resistor, current probe or EM emanations device.

Typically it is efficient against block cipher such as AES we can quickly highlight some part of the algorithm.

We can even see some capacitive charge and discharge on the lines and the amplitude reveals some information about the operand values.

There exists many approach to power analysis attack and the definition are quite flexible but the mains are :

- Simple Power Analysis SPA
- Differential Power Analysis DPA
- Correlation Power Analysis CPA

There also exists two main families:

1. Profiled: training on a classifier we know some information and what to look for
2. Non-profiled: no real knowledge about what to expect

2.2.1 SPA

Visual inspection of traces. Need the IO and some expertise.

2.3 differential side channel analysis (DSCA)

This allows to affirm or reject some hypothesis about an intermediate state of an implementation. If this kind of analysis allow to deduce some information about a secret then it becomes an attack.

For an analysis to work we need to make sure that there is a direct link between power and bit status. This is usually the case when a bit is 1 then higher power consumption and vice versa (proved).
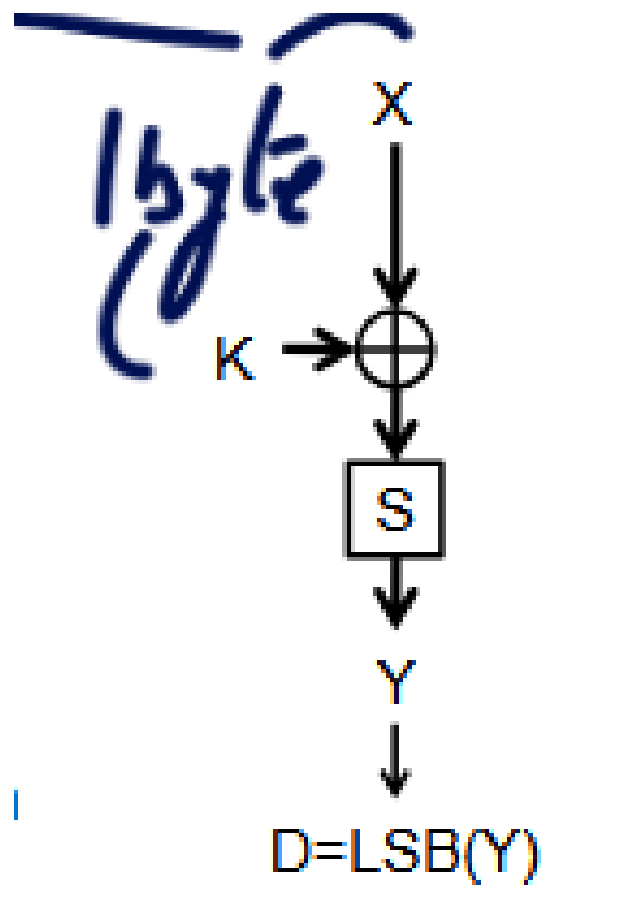
Figure 1: S-box

We use a function D to only select certain bit to simplify our analysis. The idea is like if K has this value, Y will be this which is sorted by our D function and we see if it is higher or lower in term of power consumption.

- For each possible value of K, here [0..255]
  - Compute the value of Y for each of the n inputs and apply the selection function: D=0 or D=1?
  - Group power curves in two sets: D=0 and D=1
  - Compute mean power curve for both sets
  - Compute difference of both means → differential curve

- Analyse the differential curves:
  - For the correct guess of K, the differential trace shows peaks at the point in time when the selected bit is manipulated

Figure 2: How it works

The basic idea is that most of the time, our assumption about the key is wrong and thus our categorization will also be wrong. But if our guess is correct then we will correctly put the trace in the appropriate group which will stand out from the noise.

The idea is to guess the intermediate value processed at some point in time based on our knowledge of the algorithm.

The advantage of this method is that it will reduce the influence of noise and other power consumption by activities on the chip.

2.3.0.1 How to compute DPA better   One way is to skew back the DPA around the center either by the exact or approximated formula.

It is important that those intermediate values we can target will influence the actual result or it won't help finding the key later on.

2.3.1 Correlation Power Analysis

We can also use some CPA that is more robust against noise !

3 Physically Unclonable Function (PUFs)

3.1 Introduction

We need an **entropy source** that will help us generate something truly random. There is either *static* (PUFs) or *dynamic* (TRNG).

The idea behind puf is like our own biometric data that defines our uniqueness. We want it to be unclonable so no one could try to replicate the data.

We take advantage of process variation and in this case we want this pelgrom's law that will induce more σ.

3.2 Important PUF properties and design flavors

The weak and the strong puf. Basically the weak puf will **linearly** increment the randomness while the strong will **exponentially** increase it. Both are used and have their pros and cons.

3.2.1 Weak PUF

An array of identically designed IC and each produce one or few bits . High-quality in response bits. SRAM PUF for example good key generation, device identification

3.2.2 Strong PUF

Based on mathematical operation (delay, current, voltage, . . .) and can produce huge number of response bits. Low-quality bits as they will often be correlated together !(see later) Good for IC auth.

Watch out, not feasible to clone physically but we may be able to derive a certain mathematical clone !

PUF responses are not always reproducible as noise comes into the play and overtime it may even change drastically (after accessing it too much, hot-carrier injection, . . .) ust like our fingertips.

3.2.3 Metrics

To measure the **uniqueness**, we use the Hamming distance and compare the answer from 2 PUF to the same challenge. We can also compare reliability by the same method for 2 same challenge on the same PUF.

$$PUF = \frac{HD(r_1, R_2)}{n}$$
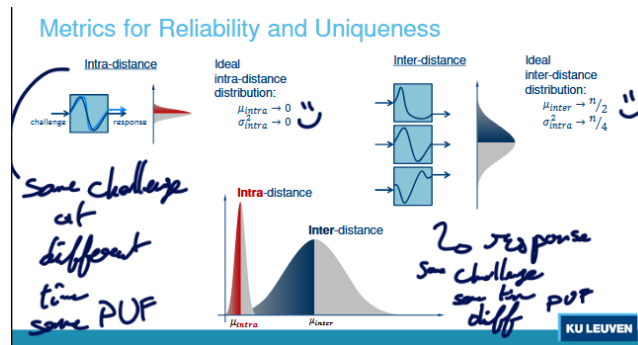
Figure 3: Ideal properties

It is quite hard to estimate the min-entropy.

$$H_\infty(X) = -log_2\left(\max_{x_i}(Pr[X = x_i])\right)$$

It is a *conservative measure of guesswork and unpredictability*. So this shows the minimum amount of randomness. So typically, if it is biased towards a certain value, we loose some randomness.

3.3 Design examples
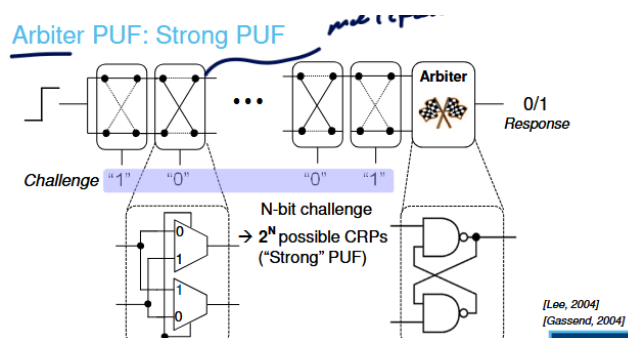
Figure 4: Arbiter for a strong PUF

As we can see, it is quite correlated ! This can be easily predicted using some NN and a trial to fix this is using XOR Arbiter PUF. The issue is the fact it will amplify the noise !

There is many example in the course **see slides**.

3.4 Improving PUF reliability

3.4.1 Temporary majority voting

To avoid to be too subjected by noise, we can reiterate the same challenge N times and do some majority voting to see what is the most probable and accurate answer. Introduces Latency and need more storage element.

3.4.2 Dark-bit masking

Mark some bits as *do not use*.

3.4.3 Burn-in enhancement

Typically, in SRAM PUFs, one side will be more often charged than the other which induces an asymetrical stress that we want to get rid of.

3.5 Applications

We can use some *helper data* to get a specific type of key. See course slide to understand the HDA method.

3.6 Summary

- IC PUFs – unique chip fingerprints
- Process variations in silicon technology
- Key properties: reliability and uniqueness
- SRAM PUF as a classic example of a weak PUF
- Strong PUFs prone to ML attacks
- Need for helper data algorithm
- Secret key generation using weak PUFs
- Lightweight entity authentication using strong PUFs – still an open question

4 Random Number Generators for Security

4.1 Why do we need random numbers?

A system is secure even if everything is known about it besides the key. So we see the importance of generating truly random numbers. Randomness is a property of data generating process/source ! It shows how unpredictable data generation is.

4.2 Mathematical background

It is all about minimizing the bias. The Shannon entropy is also important as it measures the disorder. Finally the min-entropy reveals the amount of truly random bits produce by one mean.

$$bias(X) = \left| Pr[X = 1] - \frac{1}{2} \right| = \left| Pr[X = 0] - \frac{1}{2} \right|$$

$$H_1(X) = -\sum_{i=0}^{N-1} (Pr[X = x_i]log_2(Pr[X = x_i]))$$

$$H_\infty(X) = -log_2(\max_{x_i} Pr[X = x_i])$$

Taking the min-entropy always ensure we have have 2^{-n} chance of guessing the right key at the first try. Min-entropy is always smaller than the Shanon entropy.

4.3 True Random Number Generators (TRNGs)
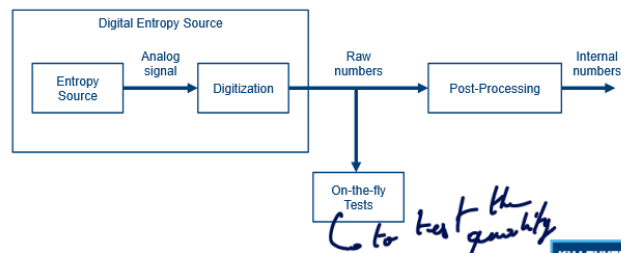
Figure 5: Typical scheme for TRNG

The "on-the-fly" tests are there to check for possible issues or biased results. Post-processing will compress the numbers to obtain a full-entropy output bits.

4.3.1 Entropy Sources
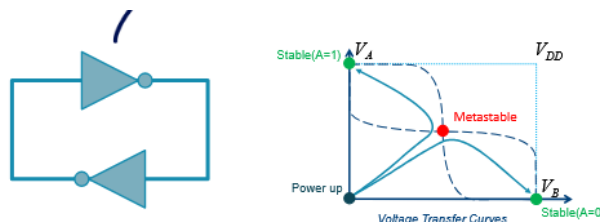
Figure 6: Metastability

4.3.1.1 Metastability   It's like rolling a ball on top of a hill and seeing when it goes left or right. But the drop of the ball is influenced by noise and the hill by the manufacturing variation.

4.3.1.2 Timing jitter   We can also inspect the jitter in a RO. TERO is not too bad.

4.3.1.3 Quantum effect   We use some photon detection circuits to check in which way the photon is fired.

4.3.2 Testing

We do some prototype testing and check long bit sequences which can be quite intensive compute wise. It needs to detect some total failure and see if some bits are stuck at a specific location. Always one full cycle of continuous tests before any output (start-up tests).

Need to be careful with some attacks and the effect could appear a lil later before any tests can check it.

4.3.3 Post-processing

We can use some *parity filter* that will reduce the throughput and will not remove the bias. But we need the bits to be totally independent from each other or their locality will increase the bias.

To remove totally the bias we can use the **Von Neumann**'s post-processing. The input bits must be independent and identically distributed (IID).

Compression is needed to increase density after post-processing.

4.3.4 Summary

- Digital entropy source

  – Produces raw bits, usually not completely unpredictable (do not have full entropy)
  – Sensitive to operating conditions (voltage, temperature, etc.)
  – Requires testing to monitor health

- Two types of testing

  – Prototype evaluation
    * Extensive statistical testing
    * Black box
  – n-the-fly testing
    * On chip
    * Lightweight, low latency
    * False positive errors

- Post-processing

  – Increasing entropy rate of the random bits ("extracts entropy" from raw bits)
  – Compression function (if there is no compression, it doesn't extract entropy)
  – XOR (parity filter) cannot remove bias completely
    * Inputs should be independent
  – Von Neumann removes all bias

           * Inputs should be independent and identically distributed (IID)

       – Cryptographic post-processing: not lightweight, offer computational security of the output bits

4.4 Pseudo Random Number Generators (PRNGs)

Uses for high throughput random number generator. It is based on an actual random numbers named the *seed*.
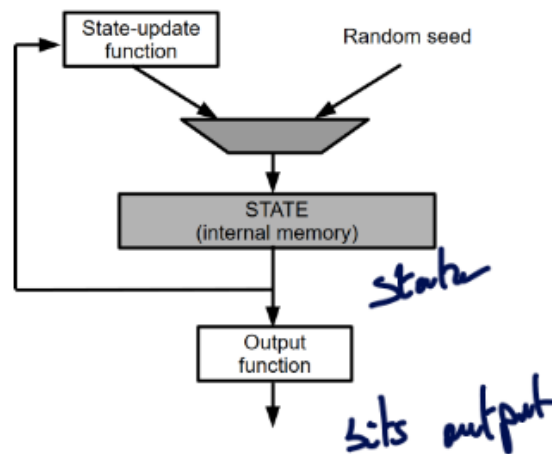
Figure 7: PRNG Architecture

If an attacker intercept a part of the sequence shouldn't be able to compute any previous or future output. This is forward and backward secrecy. We can have some *enhanced* secrecy if we can't find the future or past secrets based on **internal state** value.

4.4.1 Dual EC

There exists some non invertible algorithm usually based on the Elliptic curve !

- PRNGs are deterministic, they don't produce entropy
- Requirements:

  – Forward and backward secrecy: The attacker cannot predict future/past outputs by observing the current and past/future output
  – Enhanced forward and backward secrecy: The attacker cannot predict future/past outputs by observing the internal state

- Three main components:

  1. Internal state (memory)
  2. The state update function
  3. The output function

5 Cryptographic Key

5.1 Block cipher

Def : "A block cipher breaks up the plaintext into strings (called blocks) of a fixed length t over an alphabet A and encrypts one bloc at a time."

It will repeat on it we call this rounds and each rounds has subkey derived by key schedule.

Often, one cycle per round for HW architecture to ensure speed and throughput. On the other side we can make low area which is slower.
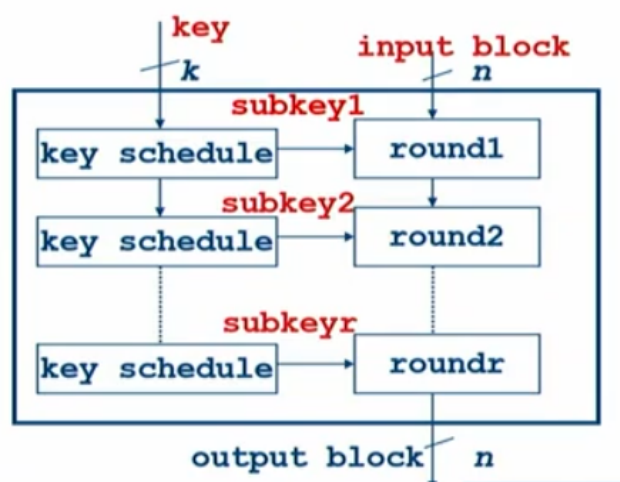
Figure 8: Block cipher example

5.2 Data Encryption Standard (DES)

It is a *block cipher* with 64 bit I/O and 56 bit key with 8 parity bits. The idea, it is iterated cipher with 16 rounds. It has influenced modern encryption even tough it is no longer considered secure as of 2004.

5.2.1 Triple Data Encryption Algorithm - TDEA

We have 3 key options :

1. K1, K2, K3 different
2. K1=K3, K2 different
3. K1=K2=K3 which makes it *backward compatible with DES*

The two-key triple DES is recommended until 2009 and three-key triple DES until 2030. Still used a lot in the payment industry.

In 2017, we limit the max b lock size to 2^{30} and disallows its usage for TLS, IPSEC, …

5.2.2 DES Feistel Structure

It is still heavily used in reality.

Figure 9: Des Feistel Structure

The encryption and decrytpion is the same function so it is super hardware efficient !

In the 16 rounds we have an initial and final permutation.

Figure 10: DES hardware

Figure 11: DEs-f function

Expansion will expand and reshuffle the bit. We then have 8 S-box with 6 inputs bit and 4 bits output. S_i non-linaear subsitution.

Figure 12: S-box

We have some needed linearity to have good and robust encryption. But too much non-linearity is costly on hardware.

We havec then the key schedule, with a 64 bits key input but we will only have 56 used

Figure 13: DES Key schedule

We rotate in those C and D register and it depends on the round we are in that decide how much we shift.
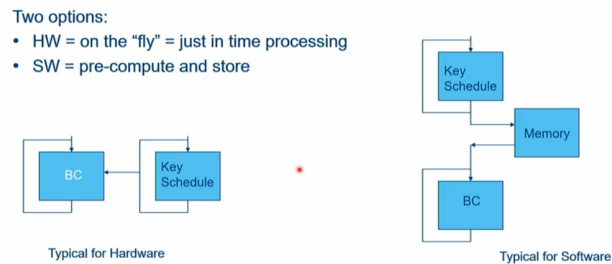
Figure 14: On the fly in hardware

The key never really change so we prefer in SW to first compute them and store in memory to easily access it.

If we want to store the key in memory we have one 128 bit key which is 1208 bits round keys so 10 rounds and initial key. But half of internet packets are only 64 bytes in length (512 bits) it doesn't make any sense anymore. It had too much overhead to load and forget keys of every users.

DES was originally developed for efficiency in HW and is quite unefficient in SW.

5.2.3 Software

We need to do many permutation of the plain text input. The issue is that it is **bit oriented** so it is annoying to use mask and move it. In ASM we don't have bit-wise operation. Cheap in HW (simple wire) expensive in SW.

The simple way takes around 300 instructions per block !

5.2.3.1 Bit slicing : alternative data representation
Each register contains 1 bit of eg 32 blocks. Block size is defined by algorithm for DES block is 64 bit. We are going to parellelize of n encryptions. Number of blocks in parallel n = width of register.
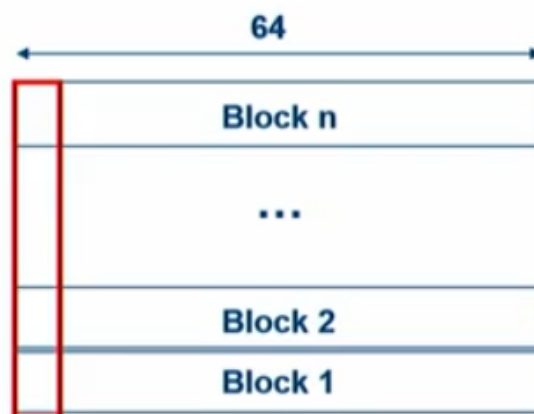
Figure 15: Operating First bits of multiple block

So in a register we have all the first or n bits of all the blocks etc. So now the CPU can be viewed as 16/32/54 one-bit parallel processors (depending of the size of the inputs). CPU is like a Single-Instruction Multiple-Data SIMD processor.

And now, for permutation we can easily copy the content of a register to another, it is no longer bit-wise operation. Easy !

5.3 Mode of Operation

The mode of operation tells what to do when we have **multiple block of data**. If we are simply following the block in the normal way, we could actually reveal the data and not encrypt it correctly. The issue is if we encrypt "A" it will have the same encrypted version for any encrypted "A".

5.3.1 Design method

Inclue modes of operation into hardware IP module or co-processor. It gives more hardware but more clean security partitioning, reduces communication overhead and traffic.

5.3.2 Cipher Block Chaining - CBC
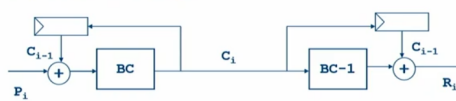
Figure 16: CBC

Error in Ci propagation over 2 blocks ! If we have a loss of block synchronization it is fatal. If we have an error in P_i we will propagate it to the other blocks. It is mostly used with encryption only for Message Authentification Encryption (MAC) generation.
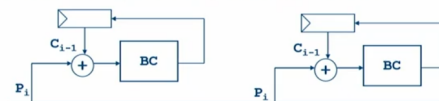
Figure 17: CBC-MAC

5.3.2.1 CBC-MAC   Feedback inhibits pipeline. But due to feedback we can't easily pipeline it. It gets even worse for triple DES. Worse for bit slicing and certain masking schemes.
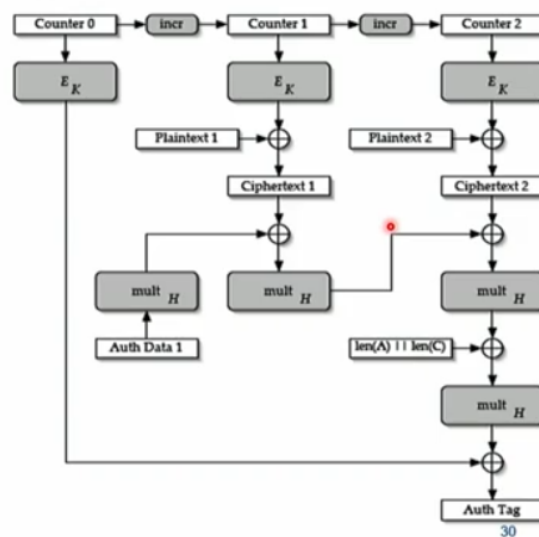
Figure 18: Galois counter mode
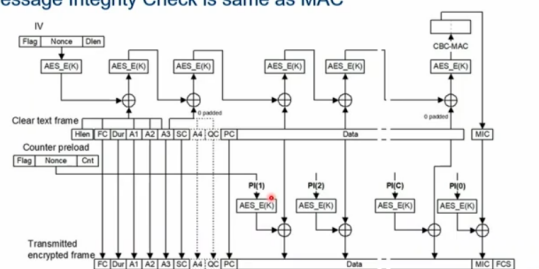
Figure 19: CCM (counter + CBC MAC) mode

5.3.2.2 Modes of operation counter   Add confidentiality and encryption.

	Sender	Receiver	Pipelining	Notes
ECB	Enc	Dec	Yes	not used
CBC	Enc	Dec	No	
CBC-MAC	Enc	Enc	No	Message authentication
CFB	Enc	Enc	No	
OFB	Enc	Enc	No	
CTR	Enc	Enc	Yes	
OCB	Enc	Dec	Yes	Privacy & MAC
CCM	2Enc	2Enc	Only CTR	Privacy & MAC

- Conclusion: most practical applications ONLY use encryption. Important for:
  - AES because encryption is more efficient than decryption (non-Feistel)
  - area constraint applications (e.g. IEEE 802.11)[32]

Figure 20: Block cipher modes of operation

The third column indicates how we encrypt or decrypt.

The counter mode, we don't have any pipelining anymore.

6 Countermeasures against physical attacks

To develop good countermeasure, we can't just focus on one area and then think it is good enough. No we need to add many layers of protection at all levels to be more resilient against attacks, we are applying it to

all level of abstraction.

There is 2 dominant philosophies:

1. **Bottom up**: if we protect a low level component then all the higher one are resilient and safe
2. **Top down**: if one component can tolerate side-channel leakage of lower components, then the lower components need no protection.

We are often limited by the "what" since we operate on input and output and also by the "how" aka the level we are working at. We can't simply change protocol and primitives as they are fixed.

6.1 The basis

To know if something is vulnerable, we need to have:

1. Sensitive data
2. Processed values that uses the sensitive data
3. Physically observable operation (power, sidechannel, ...)

For this we have multiple ways to prevent ourselves:

- Masking: decorrelate the sensitive data and the data processed
- Hiding: we are using lower SNR to make the recovery harder for an attacker
- Provable secure countermeasure: usually rely on masking but hard to validate and easy to invalidate

To know if we are at risk, we need to see if some intermediate data depends on a secret and get be guessed with some secret found by the attacker. We have a leak of information.

6.2 Countermeasures

We have 3 main CM

6.2.1 1 - Re-keying

We change the key every few often so the attacker doesn't have enough sample to find the key. We also need to make sure that the re-keying is protected and some protocols don't support this.

But it is not always enough as some protocols just need one sample to be broken.

6.2.2 2 - Masking

We make intermediate values independent of the secret. We are protecting ourselves to low-order attacks but it gets harder and harder for high-level one.

We only need to store masked sensitive data ! Masking make the data unpredictable. But can be quite challenging to implement it and not accidentally leak or have some glitches

6.2.3 3 - Hiding

We try to reduce the SNR. We can either reduce the signal using some encoding or decoupling technique or increase the noise using some additional circuity and doing some timing jitter that is controlled in software. Those effects shouldnt be undoable.

We can also make the operations always look the same or add dummy operation to not reveal the branch we are currently on.

We need to be prepared to fail as no CM is perfect.

6.3 Analog circuits and noise generators

We want to hide the power consumption, the ideal scenario would be a power supply that is isolated and cannot be seen or measured by the attacker. But we can't ofc.

6.3.1 Decoupling

We had a capacitor between V_{DD} and ground to smooth out peaks in the power consumption. But we can't perfectly smooth things out and the cap cannot be infinitely large.

6.3.2 Detach

Figure 21: Detached Power supply

The idea is to have one capacitor that charges while the other is supplying the current. Not perfect because we will see peaks in the toggling of the cap which also leak information.

So the best thing is to supply the current through a cap, discharge it completely and then charge it again so we have a constant power consumption.

6.3.3 Active flattening

We add a sensor that measures the supply current and we have a sink that mhlps having a constant power and current consumption using this sink. But this means we will have maximum current consumption all the time.

Figure 22: Active flattening

It is never perfect and even GPIO pins leak information. We are not protected against EM side channel attacks with those techniques.

6.4 Noise generators

The noise must come from an independent noise source. We gotta randomly charge and discharge a capacitor. We can also activate some unused co-processors. Both methods rely on using a source of randomness.

6.5 Masking

We know that intermediate data will be a function of a key and a sensitive data:

$$x = f(p, k)$$

But with masking, we are creating d shares which $d \geqslant 1$.

$$v = m_1 \cdot m_2 \cdot ... \cdot m_d \cdot x$$

Here, "$\cdot$" represents a type of masking. The most basic one is the **boolean masking**

6.5.1 Boolean masking

It uses a xor operation $\oplus$, it is linear which makes linear operations easy to mask. Non-linear operations are harder to mask.

6.5.2 Other types

We can also have the multiplicative one where $\otimes$ (mult in $\text{GF}(2^8)$). It is really used for S-box in AES, but we need to be careful cause we can't mask a 0 with this type of masking.

We can also use modulo addition where $+$ (addition mod n). Used for the first versions of SHA and now in Post Quantum Cryptography.

With the $\oplus$ we can recover the secret after reapplying the mask. In general, we have a vector and mask pair.

We can't do masking before going into a S-box since they are non-linear. We have to create a new S-box S'. We have to recompute based on our S-box the new S'-box based on the new mask.

7 Trusted Computing

It is the **assurance on running code**, we need to prove its identity, authenticity and secrecy. Making sure to keep sensitive data private. This is enable through **hardware support**.

Trusted Computing all rely on root of trust and chain of trust. TC solutions need to be tailored for every application.

7.1 Theory

> *Trusted system or component is one whose failure can break the security policy, while a trustworthy system or component is one that won't fail*

So a trusted device is a *dangerous* attack point. It is part of a potential attack surface. And as a hardware designer it is our task to prove security since we can't rely on further lower level supposition since we are the lowest level. That's why we are creating TPM or special operating mode TEE.

7.1.1 TCB and RoT

- Trusted Computing Base: collection of everything in a system that is trusted

    - Often implemented in a hierarchical manner where A verifies B, . . .
    - Root of Trust: the anchor of this chain

eg: when booting in windows 11, the boot ROM starts (RoT) and verifies the BIOS, then the bootloader gets verified and finally the OS. Then any untrusted applications can execute.

By using this kind of chain, we can verify authenticity and make sure no malicious actor tampered the software, . . .

But also good for Digital Right Management and make sure there is no fake copy or pirated copy that are being used.

Technique	Goal	Examples
Secure boot	Ensure only signed software is running, encrypt software binaries	every game console and phone
Measured boot	Unforgeable identity of running code	DICE
Trusted execution	Shield away code and data from untrusted OS	SGX, TrustZone

Technique	Goal	Examples
Remote attestation	Prove to other party which software stack you are running	Apple App Attest

7.2 Techniques

7.2.1 Secure boot goals

We don't want to leak the code or secrets running on device and make sure we can only execute vendor-approved code. Good for **confidentiality** and **integrity & authenticity**.

note: Secure boot is the general term, Secure Boot is the windows implementation.

Here we have on the hardware a key written with fuses and it sends its public key to the TPM unit that make sure the BIOS is not corrupted. Here the RoT is the boot ROM and the engraved key.

7.2.2 Measured boot goals

Know which code is being executed and can attest it. Good for **integrity**. Here, even if the code is not the right one, it is still allowed to run but we can check that it is not the correct one.
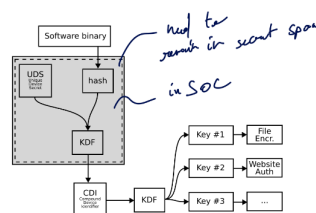
Figure 23: Device Identity Composition Engine

They grey box is done by a secured piece of code that is trusted by the boot ROM. The CDI will change at any data change in the code which creates new key making previous encrypted data unreadable.

7.2.3 Trusted Platform Module

It uses cryptography and is a form of data storage that can realize attestation. Heavily used and now standardized by the TCP.
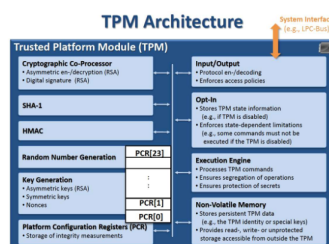
Figure 24: A. Sadeghi, used with permission

7.2.4 Trusted Execution Environment goals

It hides secrets from other system parts **confidentiality** and isolate execution from untrusted parts (like OS too) **availability**. Can be broken through leakage and side channel at the microarchitectural level.

This can be implemented like an ARM TrustZone where on top of the privileged, unprivileged mode of execution, we have a secure and non-secure mode:

Figure 25: Arm TrustZone

Also relies on code supplied by the boot firmware.

7.2.5 SGX

It creates some *unprivileged secure* islands. In other words it creates enclave shielded from the OS and uses memory encryption. It also uses attestation to prove authenticity and is implemented using some CPU microcode.

7.2.6 Remote Attestation

This is a mix of TEE and measured boot and is called **remote attestation**. We ask a server to verify some code that it is running. Then it will measure the code and will sign hash and challenge with a protected signing key.

But we need to protect some nonvolatile key storage and need a TEE implementation. We also need to be protected against some classic attack. How to anonymize prover, give keys, . . .

7.3 Conclusion

We will try to minimize the Trusted Computing Base and will move towards RoT with one single pont of failure. There isn't a one-fit-all solution, we need to create one for each threat we want to protect against.

Goal	Technique	When to Use
Only vendor-approved code runs	Secure Boot	Prevent tampering/piracy on boot
Log software state cryptographically	Measured Boot	Detect software changes; support attestation

Goal	Technique	When to Use
Isolate secrets from the OS	TEE (TrustZone/SGX)	Secure data/apps against OS or other apps
Prove software identity remotely	Remote Attestation	Enforce remote policy (DRM, anti-cheat, banking)

8 Micro-Architectural attacks and defense

Micro-architecture is something pretty subtle as it is not software nor really hardware. I sits right between the Software and ISA and the Hardware. It is something that differentiates each generation of processor between each other.

We have the Instruction Set Architecture that decodes and executes. The micro-architecture is like the secret sauce that makes everything works and that allows Hardware engineer to create and extend some functionality without redesigning ISA from the ground-up.

Specific implementation of the ISA is called the micro-architecture.

8.1 Caches

In hardware, everything is a mirage and we tend to idealize everything. We want fast and wide memory. But this is a pure dream, so we have to create *cache-levels*. Each level is a bit bigger than the previous one but also slower.

Data that was recently accessed are often physically located close to a CPU core for quick access in the future.
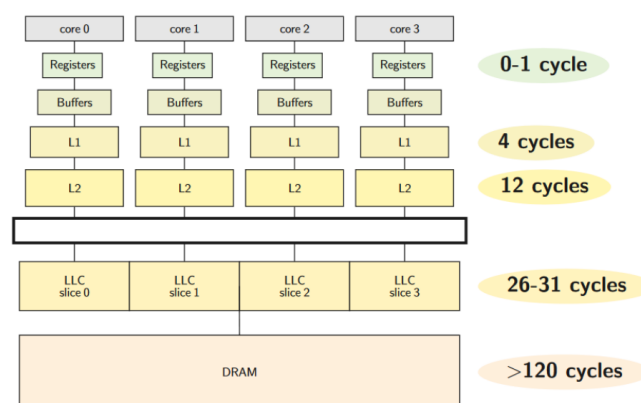
Figure 26: Cache levels

8.1.1 Set-associative cache

If we understand the architecture and the algorithm behind caches, it can potentially indicates sensitive informations to an attacker.

Typically in set-associative caches, We use the something like:

```
1  [-----tag-----:---index---:offset]
```

Where the index is the row in our cache table, the tag is there to be sure that we are accessing the right data. The offset is usually 2 bits since we often work in 32 bits system so $4 \cdot 8$ bits (we rarely access data or instructions in the middle).

So if we fill up a cache line with garbage data, we can time how long a program will run and compare various executions. This is the principle of cache attack and such attacks can be extended to page-fault attacks.

8.2 Cache side channels

To make those attacks feasible and plausible, we have to rule a set of constraint and access to make an attack count as is.

8.2.1 Flush + Reload

- Victim and attacker run on the same CPU and shares micro-architectural elements
- Attacker cannot read victim secrets directly, they do share some memory (shared library)
- Attacker can directly manipulate cache state, eg flush a cache line from the cache

The idea is to flush all the data from a cache line, especially the victim's address. Then let the victim run his codes, finally we measure the time access. If it is slow, the victim didn't access this line, if it is fast then the victim did in fact access this.

We can apply this attack model on RSA where a clear bit will simply square as a set bit will square and multiply. The idea is to wipe the access address of the square and multiply functions. Then we let a loop run and check the time access for both functions. We can then determine if the victim accessed one or both functions.

There exists some countermeasures for this as:

- Constant-time programming
- Not allowing cache state manipulation
- …

8.2.2 Evict + Reload

- Victim and attacker run on the same CPU and shares micro-architectural elements
- Attacker cannot read victim secrets directly, they do share some memory (shared library)
- Attacker can directly manipulate cache state, eg evict so remove cache line by accessing others

We first access some garbage addresses to evict a potential victim's address. The we let the victim run his codes and finally we measure the speed.

8.2.3 Prime + probe

- Victim and attacker run on the same CPU and shares micro-architectural elements

- Attacker cannot read victim secrets directly, they do share some memory (shared library). Prime cache with attacker-controlled addresses.
- Attacker can directly manipulate cache state, eg flush a cache line from the cache

Here, we will evict a cache line by using a set of addresses that map to the same cache set as the victim address. After, we will measure the speed of the cache set, if it has one or more slow access, we know the victim used this cache set.

Some counter-measures can involve:

- constant time programming
- Restricting access to timers
- Disabling cache
- Partitioning cache
- Cryptographically secure indexing function

8.3 Cache covert channels

The idea is to communicate between a sender a receiver without using an official channel. But we use the cache to send those data since the two are running on the same cpu.

For example, we have a large buffer, we can flush all of the data from the cache and then the transmitter can do a dummy access to a specific offset in this array. Then the receiver scan the access time to every a_i.

8.4 Summary Cache attack

Attack	Principle	Shared Memory Required	Flush Instruction Needed	Attacker Control Over Cache	Resolution / Granularity	Typical Use Case
Flush + Reload	Flush shared memory, let victim execute, reload, and time access	V Yes	V Yes	Direct (flush + measure)	Fine-grained (cache line)	Key extraction (e.g., RSA), spyware detection
Evict + Reload	Evict shared memory by cache eviction, then reload and time access	V Yes	X No	Indirect (evict + measure)	Fine-grained	When flush instruction is unavailable
Prime + Probe	Prime a cache set, let victim execute, then probe timing of eviction set	X No	X No	Indirect (measure set state)	Coarser (cache set)	Sandbox breaking, timing leakage
Covert Channel	Deliberate cache use to leak data between isolated processes/VMs	V Often	Varies	Cooperative sender/receiver	Byte-wise possible	Cross-domain leakage, VM escape

8.5 Transient Execution Attacks

Most modern CPUs implement some branch prediction and other type of algorithm to avoid stalls due to data dependency between different instructions. Most of those CPU's are what we call **out of order** meaning they can execute one or another instructions before another one. They will be called **transient instructions** where we have the result but we do not commit yet (no write back).

Those transient instructions will leave some micro-architectural side effects, it will change the cache state.

8.5.1 Spectre

```
1  buf_size = len(buf)
2  if (x < buf_size){
3    s = buf[x]
4    a = array[s*4096]
5  }
```

Typically we will iterate over the buffer, most of the time `x < buf_size` so it will avoid checking all the time and will execute the two instructions before. So we have some transient instructions and as seen before, the last line is a way to use covert channels.

We can access extra values without triggering a buffer overflow or we could also leak some data about the size of the buffer.

Some countermeasures can be the insertions of `lfence()` to stop speculative execution up until a certain point, making hte program slower. On the *hardware* level we can disable all speculation, prevent forwarding of transiently loaded data, roll back micro-architectural state, close cache covert channel, . . .

8.5.2 Meltdown

```
1  s = load(kernel_address)
2  a = array[s*4096]
```

Result from faulting instruction may have already been forwarded. Executing the array access affects the cache and an attacker can infer value of s from cache state.

There exists some software and hardware countermeasures for it such as:

- Software: Kernel Page Table Isolation

    - Don't map kernel pages in userspace

- Hardware: Don't forward values from faulting instructions

8.6 Rowhammer

This type of attacks is even more low-level as it is linked with the physical way that data is stored in RAM. In a RAM stick, each data is stored on a capacitance that is accessed through the word- and bitline. This cap needs to be charged and discharged quite often.

Reading in DRAM is destructive and requires to refresh the value. We use a *row buffer* to store the row that is being currently read.

BUT, reading all the time can disturb the neighboring rows. We can witness some bitflip.
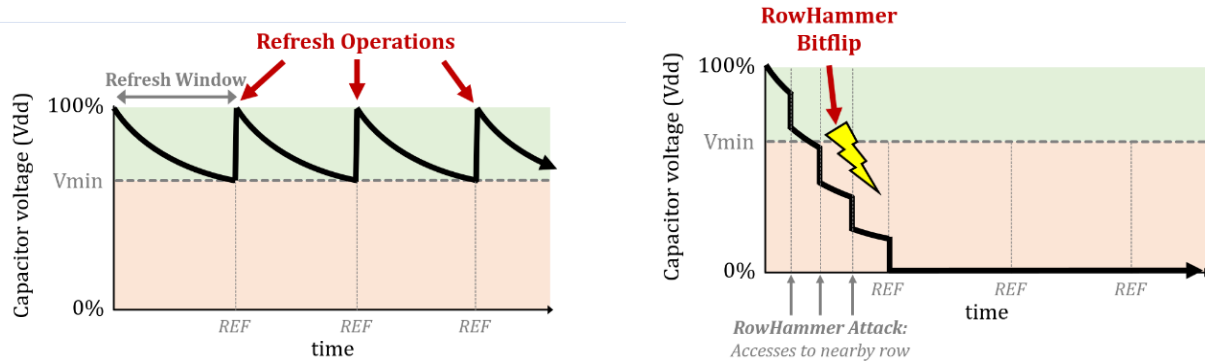
Neighboring rows loose a tiny amount of charge which causes to flip.

8.6.1 Countermeasures - DDR4

Target Row Refresh (TRR), we keep a counter for each row and if it exceeds a certain value we also refresh neighboring rows. But it is quite not feasible to implement such counter. So we have a limited number of counters per bank. So that makes DDR4 still vulnerable to rowhammer.

8.6.2 Countermeasures - DDR5

We have on die ECC, less refresh intervals so less time to hammer the rows. But it as been proved on AMD processor that we can do some rowhammer.

8.7 Conclusions

Correct execution doesn't mean it is safe. We have to often reduce performance to enhance security since performance can introduce vulnerabilities.

9 Introduction to Software Security

9.1 Introduction and Motivation

9.1.1 Evolution of the cyberspace

We have more and more IoT devices so more and more edge devices connected to the internet. The code are becoming more and more sophisticated and we are running even more programs in parallel. This all involves billions of users who rely more and more on it.

Raise the security risk from this increase.

9.1.2 Cybersecurity

Study of the general problem of:

- Maintaining good properties of Internet-connected systems in the presence of intelligent adversaries trying to break those properties

It is still a pretty vague definition, we instead will accept the definition of "avoiding bad things to happen". So we have to recognize and respond to it.

9.1.3 Past incidents

9.1.3.1 Malware

- Virus: piece of code that can infect other programs. It can then replicate through different medium.
- Worm: self-replicating virus

Now, most of the malware are developed by:

- Criminals for DDOS, botnet (rent them), crypto-mining, …
- Nations for collecting intelligence and do some cyber-offensive operations

A common attack vector for Nations is to use some supply chain attacks where a third party compromise an entity.

9.1.3.2 Jail Breaking   To do some escalation of privileges on a closed device. To customize the device, to bypass the DRM, …

9.1.3.3 Online scams

- Phishing: fake email that look legitimate
- Catfishing: impersonating someone (les brouteurs)

9.1.4 Underlying causes

There is often 2 weak links:

1. People: easy to manipulate, forget password, reuse the same all the time, security configuration, … We can even do some *spear-phishing* to target a specific person that can be at the foundation of the company's security, …
2. Software: some vulnerability in the software, … over 100 000 CVE registered.

9.1.4.1 0 day vulnerability   Vulnerability that the manufacturer is not aware of. Highly powerful vulnerability. Valuable to hackers, state, law enforcement, …

All those vulnerabilities can be also researched and found. We can either report it as part of a bug bounty program or sell it as a weapon to various entity.

9.1.4.2 Legality ? We can do some legal ethical hacking where it is not problematic. But we can still sell it which is like a weapon but seems well tolerated in the eu.

9.2 Low-Level Software security

9.2.1 Threat model

Before diving any further, we need to set the attacker and System model:

- **System:** software written in a C like language compiled to a modern processor
- **Attacker:** attacker can interact with the programs with input and sees the output. He also knows the source code and understand it and can execute it.

9.2.2 System model

It will be system dependent so the attack may vary from system to system but the underlying attack remains the same. So we will build an abstract model of the machine

9.2.2.1 Abstract model We have a memory part that is *byte-addressable* ranging from a minimum and maximum either with virtual or physical memory.

It also has a CPU that possess a Program Counter, stack pointer, register, ... with an ISA where the code is in byte in sequence. We also assume those data are **little-endian**.

Figure 27: Little-endian

We assume the code to be like a C program where we have int, char, ... and a table or vector is simply a pointer to the start of this table. We have some simple IO and other libraries and we have some private and global variable.
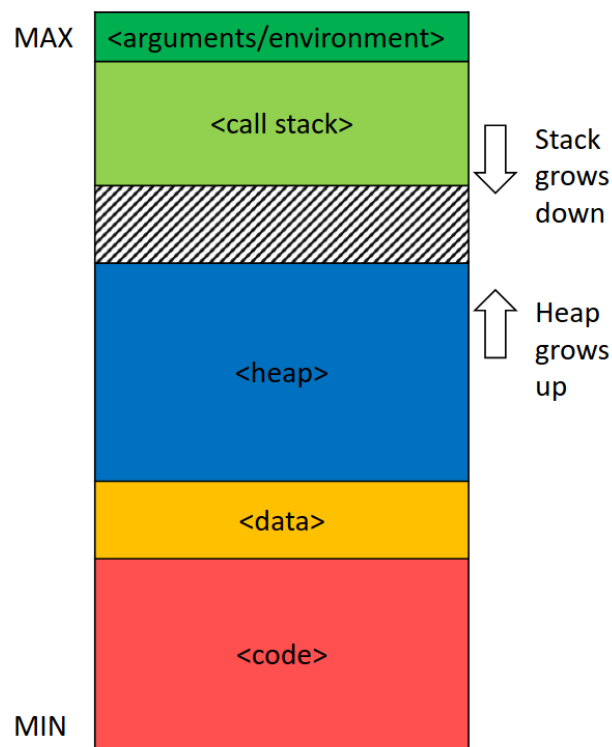
Figure 28: Stack and heap in linux

Quick reminder:

- Local variable: grow on the stack, parameters, …
- Global variable: in the data segment
- Dynamic variable: in the heap and is handled by malloc or free

We use pointers where pointers arithmetic is possible. The code for the function sits in the code and we compile the code separately. We also have some shared libraries that can be dynamically loaded.

9.2.2.2 Memory management vulnerabilities C offers a lot of flexibility and let the programmer to allocate and re-allocate variable easily. But it needs to be well used and not using variables that have been de-allocated or accessing value outside of a buffer. Usually, programs don't do checks for speed and efficiency reason but the program will run in an undefined state.

9.2.3 Attack scenarios

The main idea is to feed the program some wrong data that will take advantage of this undefined state of the program to execute or run other things. Or, since we know how is the program written we could simply trigger a specific part of the program without doing anything not permitted.

9.2.3.1 1: Call stack smashing The simplest type of attack where we will make a buffer overflow and overwrites the return pointer. We can re-write this return pointer with a shellcode.

So typically we will load in the buffer the shellcode we want, and we will put as the return address the address at the start of our malicious payload.

9.2.3.2 2: Code reuse attacks   It is similar to the previous example but here we will replace the return pointer to a specific part of the program, for example forcing to execute a function we couldn't reach but has some secrets.

9.2.3.3 Return Oriented Programming or ROP   this is a more modern idea, instead of using some return into functions, we chain *gadgets*. A gadget is a small piece of machine code that ends with a return.
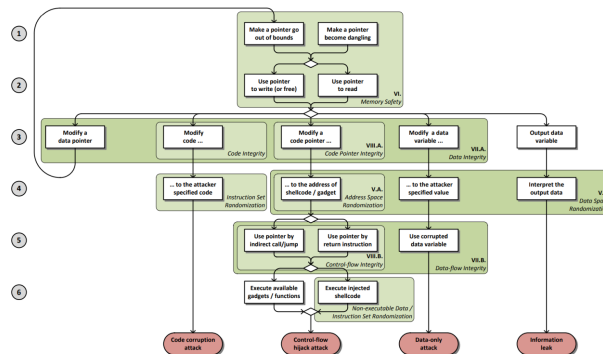
Figure 29: Basic Memory Exploit

9.2.4 Mitigating Attacks

The first defense line is the the hardware, system, compiler, ... But those protections are simply partial.

9.2.4.1 1: Canaries   We can put a small value before the return pointer and we check before return that this value was not erased or modified. A modified value is an indicator of a modified return address pointer.

9.2.4.2 2: Non-executable data   Make the stack and heap non-executable and mark the code part as non writable. This can be a good counter-measure but doesn't protect to data-only and code-reuse attack. But this could also break some legacy applications.

9.2.4.3 3: Randomize address   Use some Address Space Layout Randomization (ASLR) is a good and simple counter-measures as it significantly raise the bar for attacker who relies on exact memory address.

9.2.4.4 Other mitigations   Those automatic countermeasures are simply efficient "mitigate-the-exploit" approaches are just stop-gap measures.

9.3 Conclusions

It has been an attacker-defender infinite race against each other. Due to legacy and old codes, it has been hard to deploy solutions against the memory safety issue.

But there is some hope with some safe compilation of C but also other programming language to build system such as Rust.

10 Students Presentation

Here is the compilation of all students questions and prepared answer

10.1 2 - Low Cost and Precise Jitter Measurement Method for TRNG Entropy Assessment

We have issues of drift due to jitter, duty cycle can slightly change. It is pretty volatile cause the source of noise varies. To measure the jitter, we will avoid using probes etc since it can be worse. Use a counter:
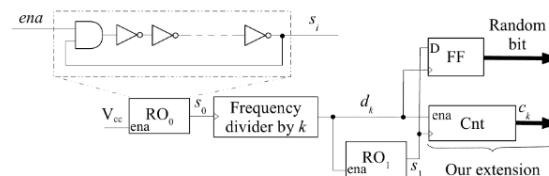
Figure 30: Counter to check jitter
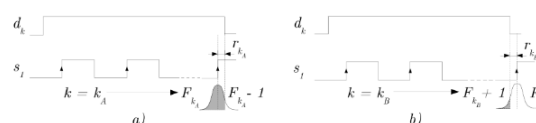
New method proposed:
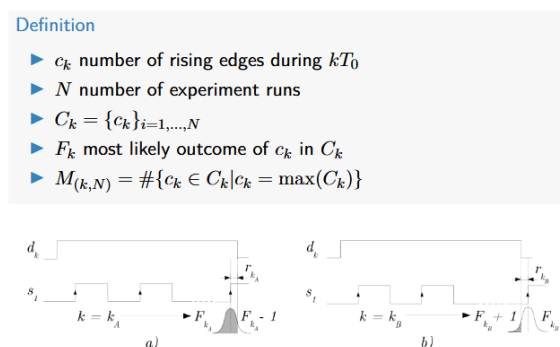
Figure 31: New method

We most likely observes the k clock cycles as expected

10.1.1 Theoretical error

$$\varphi_0 + (F_{k_A} - 1)T_1 + r_{k_A} = k_A T_0.$$

$$c_k = F_{k_A} \Leftrightarrow q_{F_{k_A}} \leq r_{k_A},$$

$$Pr(c_k = F_{k_A}) = Pr\left(q_{F_{k_A}} \leq r_{k_A}\right).$$

Figure 32: 2 cases

F_k is the most probable outcome.

10.1.2 Qestions :

"Why is a short accumulation time a desirable property of a jitter measurement technique?"

10.2 3 - Contactless Electromagnetic Active Attack on Ring Oscillator Based True Random Number Generator

It is about TRNG that are based on ring-oscillator. It generates a jittery clock. The paper will focus on attacking the source of attack, it is an active attack.

Inject EM harmonic signal to bias that source of entropy. They us a micro-probe for it. The TRNG is implement on a FPGA board.

10.2.1 Injection Platform

- Power injection chain

    – Inject wave of different power, with a frequency close to the RO (300-325 MHz)

- Control chain

    – Check when shielded from EM. Try with and without EM injection. Store and compare TRNG output bitstream.

- data acquisition chain

They use some powermeter and oscilloscope.

They used DFTRi (discrete fourier transform ratio). They check the difference between the power at the injected frequency and the power of the output RO frequency. Higher DFTRi means higher effective attack.

OFC, the higher power we inject the more effective it is.

We can use some circuitry to make it dynamic attack.

10.2.2 Question

"The attack demonstrates that ROs can be locked onto an injected frequency, leading to a biased TRNG output. How can mutual information be used to assess the effectiveness of this attack ?"

11 License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0).

You are free to:

- **Share** — copy and redistribute the material in any medium or format.
- **Adapt** — remix, transform, and build upon the material.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

For the full legal text of the license, please visit: https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode

11.1 Modification

To contribute to this work or any other one from this project please find more information at the Github repository.