Lecture 17: Hardware Trust

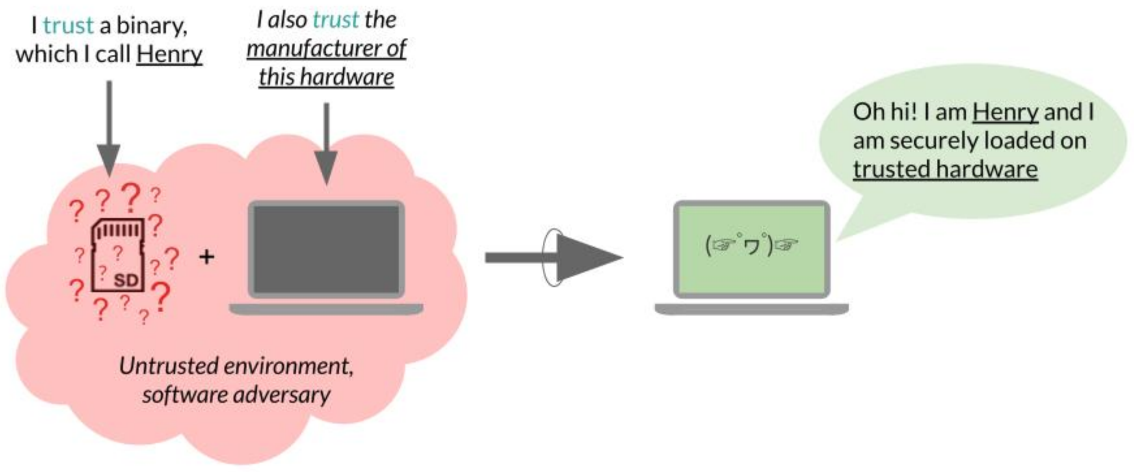
When we interact with computers, how do we know anything is what it says it is? We would like to have systems that can provide these functionalities:

Gatekeeping: Restricting the software that can run on a processor/device/chip. This is the secure boot from Lecture 16. Example: Intel SGX microcode patches, industrial microcontrollers.

Certified Execution: Processor/device/chip guarantees that particular software was run correctly and to completion. This is the measured boot from Lecture 16. Examples are Google OpenTitan, Intel SGX attested execution.

Intellectual Property Protection: Software only runs on a particular processor/device/chip. Example: Game controllers.

We will now look at a specific scenario of certified execution/measured boot and detail how we can build a system that can authenticate itself to a user.


I trust a binary, which I call Henry
I also trust the manufacturer of this hardware
SD
Untrusted environment, software adversary
Oh hi! I am Henry and I am securely loaded on trusted hardware
(☞ﾟヮﾟ)☞

Goal: Guarantee integrity for loading a boot image in an untrusted setting, given pre-existing trust in HW, manufacturer, desired boot image.

Threat Model:

Unconditional trust:

- Manufacturer's **keys**, trusted **HW**

- Desired **boot image** (key privacy)
- … and **you**! (the first party)

Adversary: boot a different, malicious binary, interact with it
… but not tamper with the hardware
… or extract its secret key
… or deny service

Steps in Measured Boot

1. *Manufacturer **M** securely <u>generates</u> PK_M/SK_M, <u>publishes</u> PK_M and <u>stores</u> SK_M.* Manufacturer is a Certification Authority.

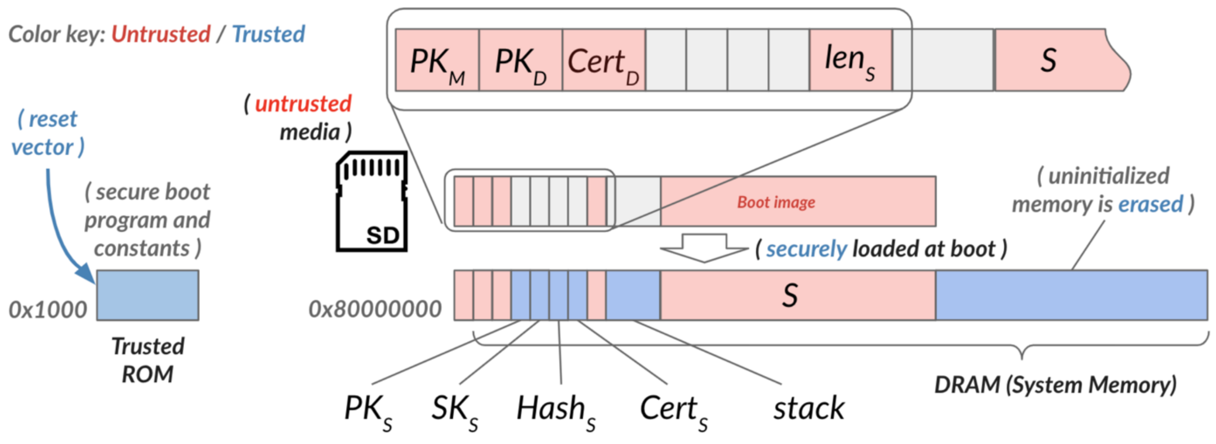
2. ***M*** builds trustworthy device D with keys. **D** has some properties:
   a. Trustworthy **keys, PK_D/SK_D, SK_D** is secret
   b. No back doors
   c. A trusted **ROM** (Read Only Memory provides integrity not privacy)
   d. First instruction integrity

   D *exposes* SK_D for use via a special interface, which is hidden from normal software.

3. ***M** certifies* PK_D, i.e., Sign(SK_M, PK_D) = $Cert_D$. Proves (under trust assumptions) that a device **D** with access to **SK_D** is manufactured by **M**.

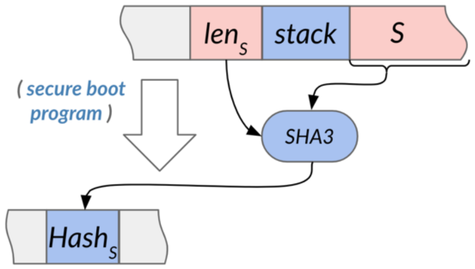
4. ***D*** executes from a root of trust at reset. ***D*** <u>must</u> guarantee first instruction integrity
   a. **Erase** (or re-key) memory
      … secrets from previous boot
      (erasing GBs of DRAM is very slow!)
   b. **Load** an untrusted binary ***S***
      … assuming it has certain structure
      (size, reserved space for keys, etc.)

   The processor is wired to execute at address **0x1000** at reset, which can correspond to a trusted ROM (a read-only memory and a part of the TCB). Via the code in this trusted ROM, the processor initializes memory with a software image, sets registers to known values, and transfers control to program that we wish to load, namely, ***S***, in system memory. The measured-boot protocol extends the boot ROM to perform additional operations described in subsequent steps 5-8, to set up the following memory layout (pink is untrusted because it comes from the outside, blue is trusted because it is computed within the chip):

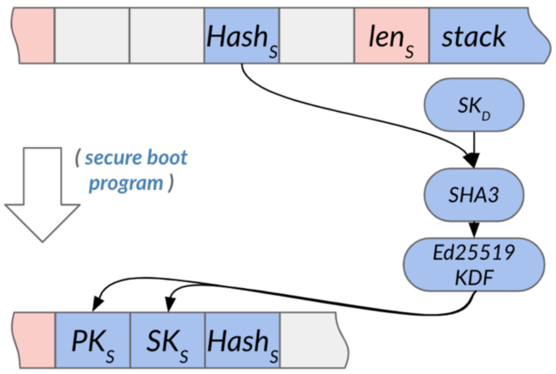


After reset, the machine executes instructions at address **0x1000**, here referred to as the *reset vector*. Before transferring control to the loaded image (***S***), the reset vector cleans up any private state left after the execution of the measured boot.

5. The bootloader measures the initial state of the system software to be loaded. In order to authenticate ***S***, the protocol *measures* it by computing a hash of a range of addresses in system memory where **S** is declared to reside. The software image to be loaded via a bootloader must correctly specify the size of ***S*** (len_s Bytes), ***S*** itself, and correctly place these at their appropriate addresses.



6. The bootloader endows ***S*** with unique cryptographic keys rooted in its measurement. Derive a cryptographic key pair (***PK_S***, ***SK_S***) for ***S*** in such a way that loading another ***S*** or performing the same bootloading procedure on another ***D*** would yield different keys. In order to achieve this property, we use a *key derivation function* (KDF) to derive the keys from both ***SK_D*** and the measurement of ***S***, as shown below.



Note that executing this code *will* leave side effects in the machine (buffers, stack frames, and micro-architectural effects, depending on the threat model), and we must take care to clean up before delegating control to untrusted software.

7. The bootloader certifies the keys of the software to be loaded. The unique keys created for ***S*** are meaningless unless they are cryptographically linked to the measurement of ***S*** and ***D***, which is in turn linked to the trusted ***M*** via a certificate. Without this, it cannot be shown that the keys have anything to do with ***S*** executing in a trusted environment, and therefore any proof of ownership of ***SK_S*** is meaningless. In order to

certify ***PK_S***, we sign the message (***PK_S***, Hash(**S**)) with ***SK_D***, creating a certificate ***$Cert_S$*** that can convince a third party that PKs corresponds to a specific post-boot environment ***S*** loaded by a specific trusted processor ***D***, which is in turn manufactured by ***M***.

8. Now we are ready to clean up and boot! The measured-boot implementation must take care to sanitize the stack before delegating control to the binary it has loaded. the post-boot environment is expected to safeguard its unique key ***SK_S*** and this can be challenging.

9. The post-boot environment has two certificates: (***$Cert_D$***, ***$Cert_S$***). Attestation of this environment includes a nonce in order to make sure a copy of an honest attestation cannot be reused by a malicious system after the fact. The trusted first party selects a nonce and has the well-known and trusted ***PK_M***. The system ***S*** creates an attestation data structure with the nonce, signs it with ***SK_S***, and sends the entire attestation to the first party. The first party verifies the attestation by verifying ***$Cert_D$*** to check if ***M*** endorsed ***D*** with public key ***PK_D***. They also verify ***$Cert_S$*** to check whether device ***D*** endorsed ***S*** with hash ***$Hash_S$***, which matches a trusted hash, and public key ***PK_S***. Finally, they verify the signature over the attestation data structure, showing the prover had access to ***SK_S***, and used the correct nonce. Key agreement can follow a successful attestation to obtain a secure channel between the trusted first party and the prover.

Gatekeeping: restricting which software the system may boot

In a setting where the space of software the system is allowed to run should be restricted (systems that expect correct software to prevent injury, for example), the measured-boot protocol presented above is insufficient. Indeed, *any* software will boot, with distinct hashes, keys and certificates. In a high-assurance setting where the system is able to perform trusted tasks *without* remote attestation, the software must be attested to locally, for example by aborting the bootloader if the software is not trustworthy.

The simplest and least flexible implementation of a gatekeeping secure bootloader extends the measured boot protocol to *panic* (stop with a terminal error) if ***$Hash_S$*** does not match a hard-coded constant. This only allows one specific ***S*** to be loaded at boot. Essentially, manufacturer has a whitelist of allowed software.

Clearly, this is not ideal: the hardware likely resists alterations, meaning a system with a hard-coded expected measurement cannot be amended by the manufacturer to address errors, changing requirements, or to increase security parameters. Instead, the hardware can hard-code the public key corresponding to a signing authority and use it to require a signed certificate for the boot image. Finally, the manufacturer can delay and delegate the white-listing of allowed software to trusted vendors by requiring ***$Hash_S$*** to be *certified* either by ***M*** or by a party endorsed by ***M***. This, while still very simple, approaches the functionality of a typical bootloader for a commercial computer system.

Reducing trust in the manufacturer

Until we figure out a way to have a piece of silicon cryptographically measure itself, we don't have much hope to authenticate hardware without some blind trust in the manufacturer and their supply chain. Given the staggering complexity of modern hardware design and manufacturing, we must trust the *integrity* of the design and manufacturing process. If a manufacturer says "this is the chip, and here are the security invariants it upholds", we pretty much have to take their word for it and implicitly trust all the third parties trusted by the manufacturer.

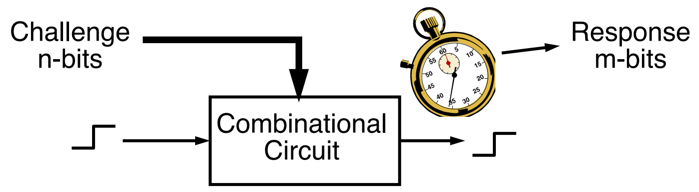
We can, however, expand the space of dishonest behaviors we can tolerate by the manufacturer without compromising the security guarantees of system. One such opportunity is reducing trust in the manufacturer's handling of device keys. Because a manufacturer is expected to endow each system they build with a unique cryptographic key pair, they may choose to go further and retain a copy of the system's private key. In order for the system's security invariants to remain viable, the manufacturer must be trusted not to reveal this key, including through malice, negligence, subpoena, social engineering, etc. It is therefore clear that the manufacturer may possess sensitive information that forever remains a critical part of the system. Consider an "*honest but curious manufacturer*" that follows the prescribed protocol and correctly manufactures and provisions the trusted computer system, but is expected to be "curious" and attempt to learn secret information. The manufacturer is not required to know SK_D in order to manufacture the device — only PK_D to endorse it. A Physical Unclonable Function (PUF) can measure manufacturing variation at boot to seed a repeatable but unique device key pair $\{SK_D, PK_D\}$ without storing the keys in any non-volatile memory, and outputs PK_D for the manufacturer to endorse. If correctly manufactured, ***M*** would have no special access to SK_D, reducing secret information outside the processor package to the manufacturer's root key SK_M alone.

Physical Unclonable Function (PUF)

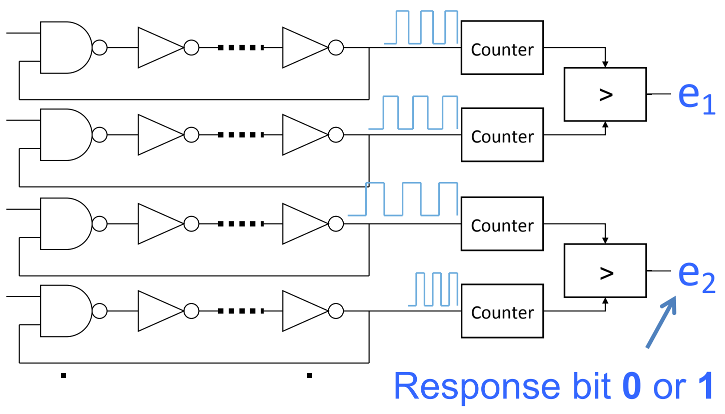
Goal: SK_D that is *hard to extract* via a physical attack, and SK_D that is *unknown* to ***M***.

Physical Unclonable Functions (PUFs)

- **PUFs are a silicon biometric**
- Because of random process variations, no two Integrated Circuits even with the same layouts are identical
- Delay-Based Silicon PUF concept (2002)
  - Generate keys from unique delay features of chips

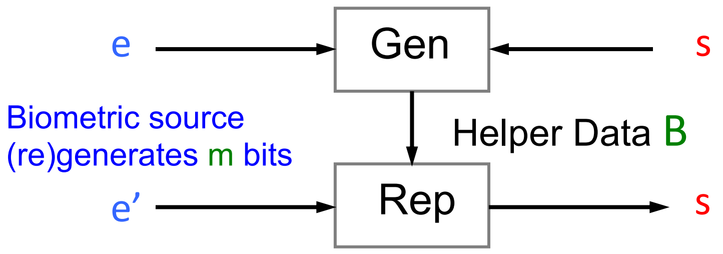


Ring Oscillator PUF



Problem: Response bits can flip **0** ⇄ **1** with temperature variation, environmental noise if compared counter values are close to each other

Fuzzy Extractors Produce Stable Keys



- Public Helper Data B leaks information about e, s
- Information-theoretically speaking, number of remaining secret bits = |s| - |B|
- Problem: error rate ↑ ⇒ |B| ↑ ⇒ secret bits ↓

Learning Parity with Noise (LPN)

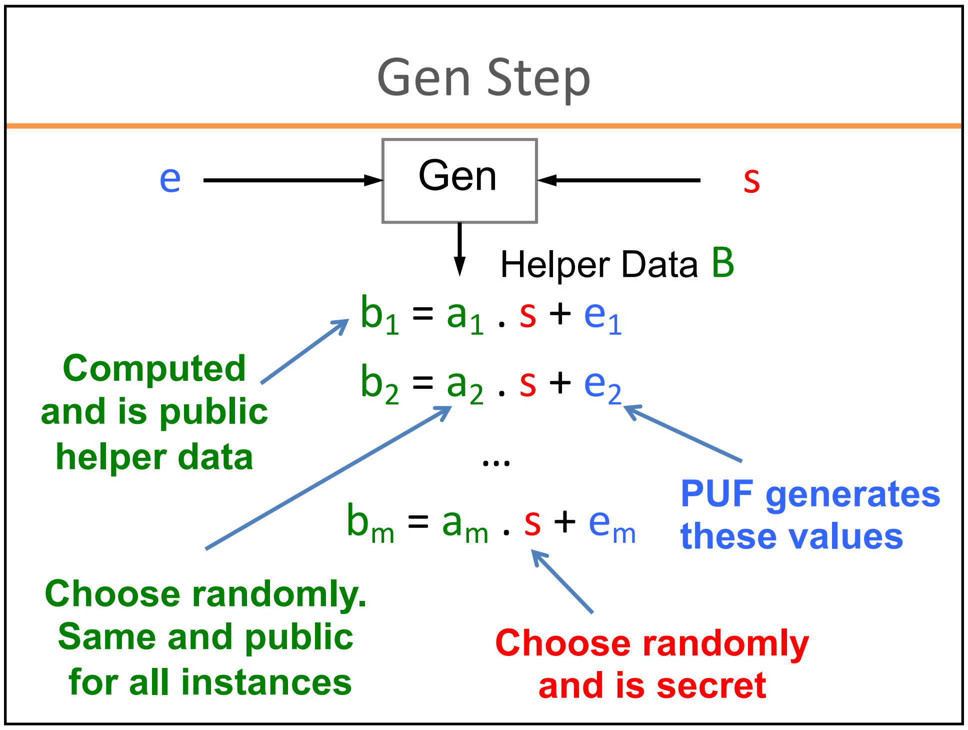
$$b_1 = a_1 . s + e_1$$
$$b_2 = a_2 . s + e_2$$
$$\dots$$
$$b_m = a_m . s + e_m$$

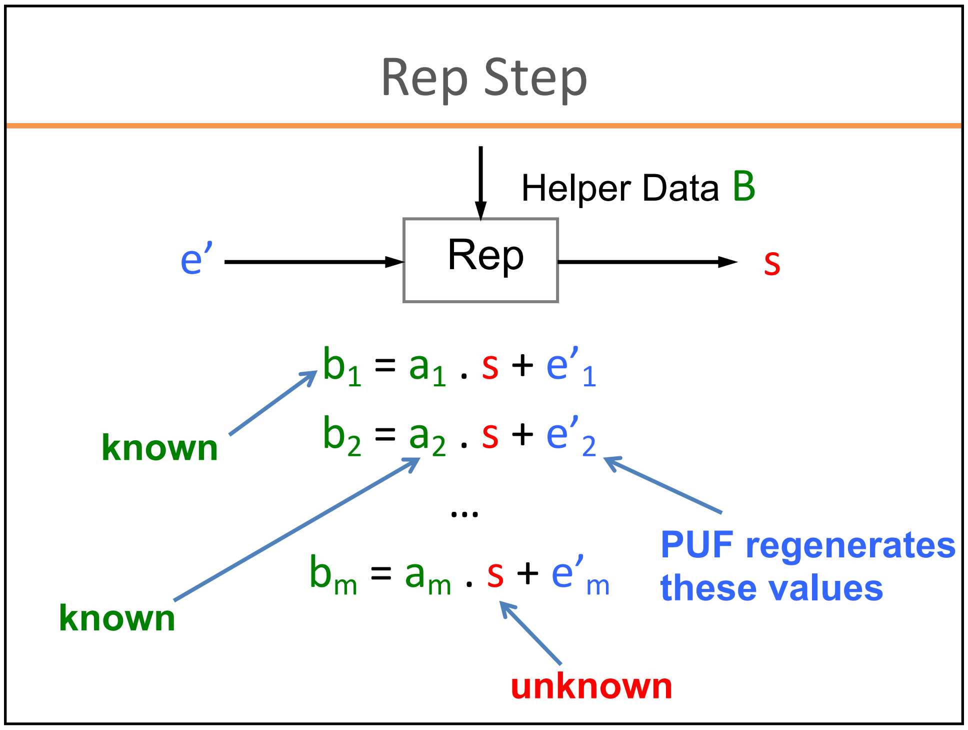
All operations mod 2

s secret, a_i, b_i public, e_i hidden independent noise

a_i, s are n-bit vectors, b_i, e_i are bits

Hard to discover s given a_i and b_i
for any $m > n$ for any non-zero noise level


Gen Step
e
Gen
s
Helper Data B
$b_1 = a_1 . s + e_1$
$b_2 = a_2 . s + e_2$
...
$b_m = a_m . s + e_m$
Computed and is public helper data
PUF generates these values
Choose randomly. Same and public for all instances
Choose randomly and is secret


Rep Step
Helper Data B
e'
Rep
s
$b_1 = a_1 . s + e'_1$
$b_2 = a_2 . s + e'_2$
...
$b_m = a_m . s + e'_m$
known
known
PUF regenerates these values
unknown

Rep Step

$$b_1 = a_1 \, . \, s + e'_1$$

$$b_2 = a_2 \, . \, s + e'_2$$

$$\dots$$

$$b_m = a_m \, . \, s + e'_m$$

known

known

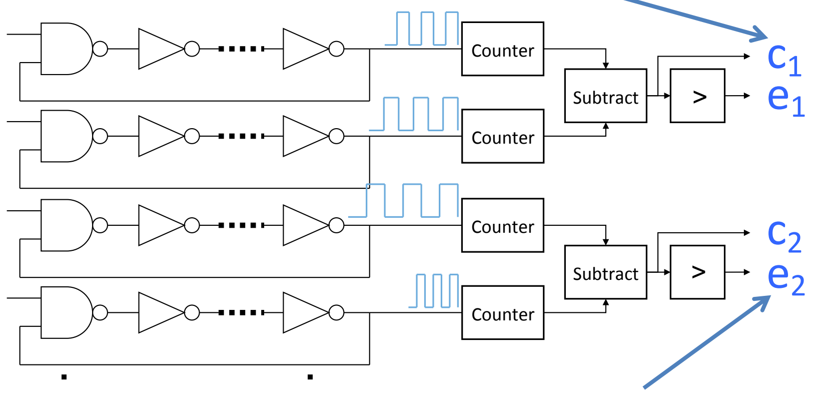
PUF regenerates these values

unknown

- Problem: e'_i values not the same as e_i values
- LPN is hard even for small amount of noise

Ring Oscillator PUF

Confidence information: stability of the bit



Response bit 0 or 1

Confidence Information

$$b_1 = a_1 . s + e'_1$$

…

$$b_m = a_m . s + e'_m$$

Need only n out of m e'_i values to be correct to solve for s

Can use $|c_i|$ confidence information associated with each e'_i value to find n correct values!

($|\,|$ denotes absolute value of counter)

Key Extraction

$$b_1 = a_1 . s + e'_1$$

$$b_5 = a_5 . s + e'_5$$

…

$$b_{m-1} = a_{m-1} . s + e'_{m-1}$$

Set m to be $\boldsymbol{K}n$, and choose the most stable n bits out of the m using confidence information c_i

Obtain s through Gaussian elimination

K grows with error rate but doesn't affect security!

Adversary doesn't know e_i/e'_i/c_i – faces LPN