Secure Embedded Systems: A Software-Hardware Symbiosis

Patrick Schaumont
ECE Department, Virginia Tech

2 April 2010
CS Department, Virginia Tech

VirginiaTech
Invent the Future

Embedded Security? Where?

Wireless keys and access control

Authenticate

Embedded Security? Where?

Electronic Money

$$$

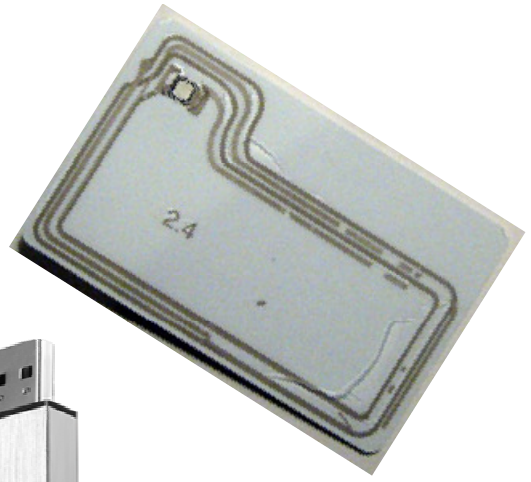
Request
+ ***Signature***

Approval
$$$

Embedded Security? Where?

Protecting Bits at Rest

Integrity



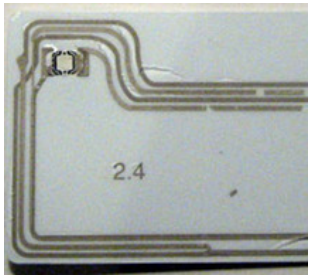
Confidentiality

Stored Secrets

- **Integrity** — **(key-less) hash**
- **Confidentiality** — **Symmetric-Key**
- **Authentication** — **Symmetric-Key/ Public-Key**
- **Non-repudiation (signing)** — **Public-Key**

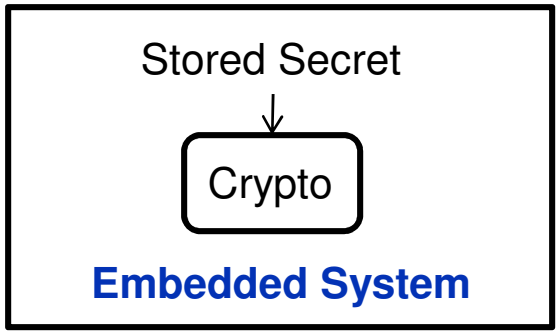
Embedded Security relies on *stored* secrets

Common Technologies

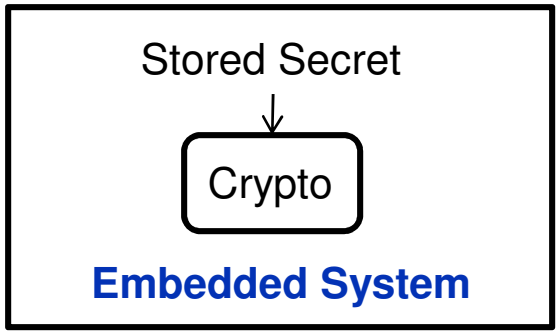


Architecture		**Dedicated Hardware**	**MicroController** 4 - 8 bit	**MicroController with Accelerator Hardware** 16 - 32 bit
Memory	Program	—	Several Kbytes	Several 100's Kbytes
	Data	100's bits	100's bytes	Several Kbytes
MOPS		100's KHz	1 MHz	50 MHz
Power		30 µW	5 mW	100 mW

Embedded Security Challenges



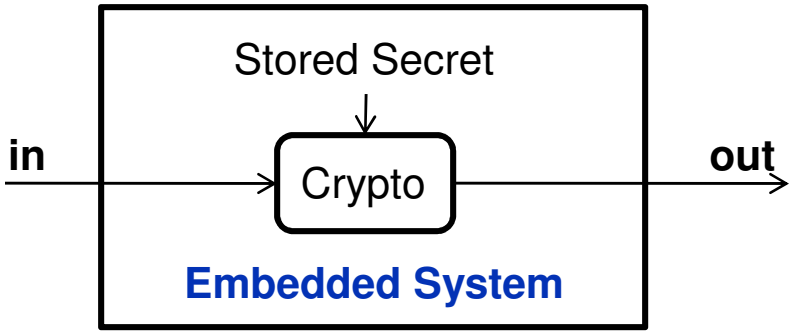
Embedded Security Challenges



Challenge #1: Dealing with Resource Constraints

Signing ECDSA secp160 p163	**Dedicated Hardware (Low Power)**			
	0.13mm CMOS 500KHz 18KGates 400mW	Sig Generation	**0.41s**	[Gaubatz 05]
	Micro-Controller Software (Sensor Node)			
	AVR ATMega128 8MHz	Sig Generation	**2.00s**	[Liu 08]
	Workstation Software			
	Intel Core 2 Q6600 2.4GHz	Sig Generation	**1.36ms**	[EBACS 10]

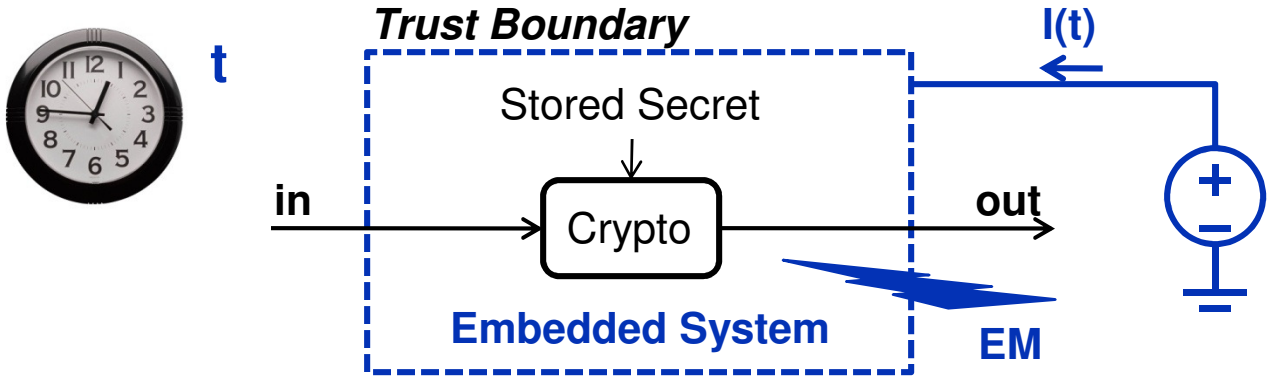
Embedded Security Challenges



Challenge #2: Dealing with Implementation Attacks

- **Traditional cryptography assumes black-box implementations: attacks should *only* consider input/output data.**

Embedded Security Challenges

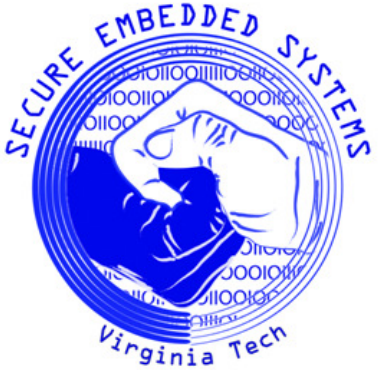


Challenge #2: Dealing with Implementation Attacks

- **Traditional cryptography assumes black-box implementations: attacks should *only* consider input/output data.**
- **Secure Embedded Systems are gray-box systems: their implementation characteristics (power dissipation, execution time, radiation, ...) can be observed**
- **Implementation attacks exploit features of the physical implementation**

Our Research

- **How to implement trustworthy secure embedded systems**
  - **that can thwart attacks?**
  - **that are efficient?**

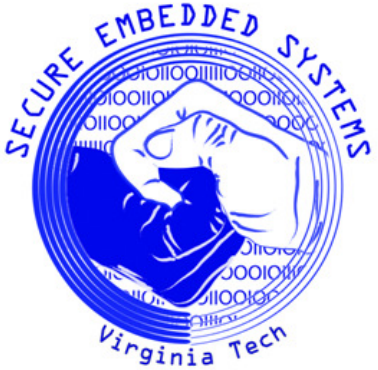


<u>Two examples of ongoing projects</u>

1. **Preventing Implementation Attacks on Software**
2. **Chip-Unique Binding of Software and Hardware**

Our Research

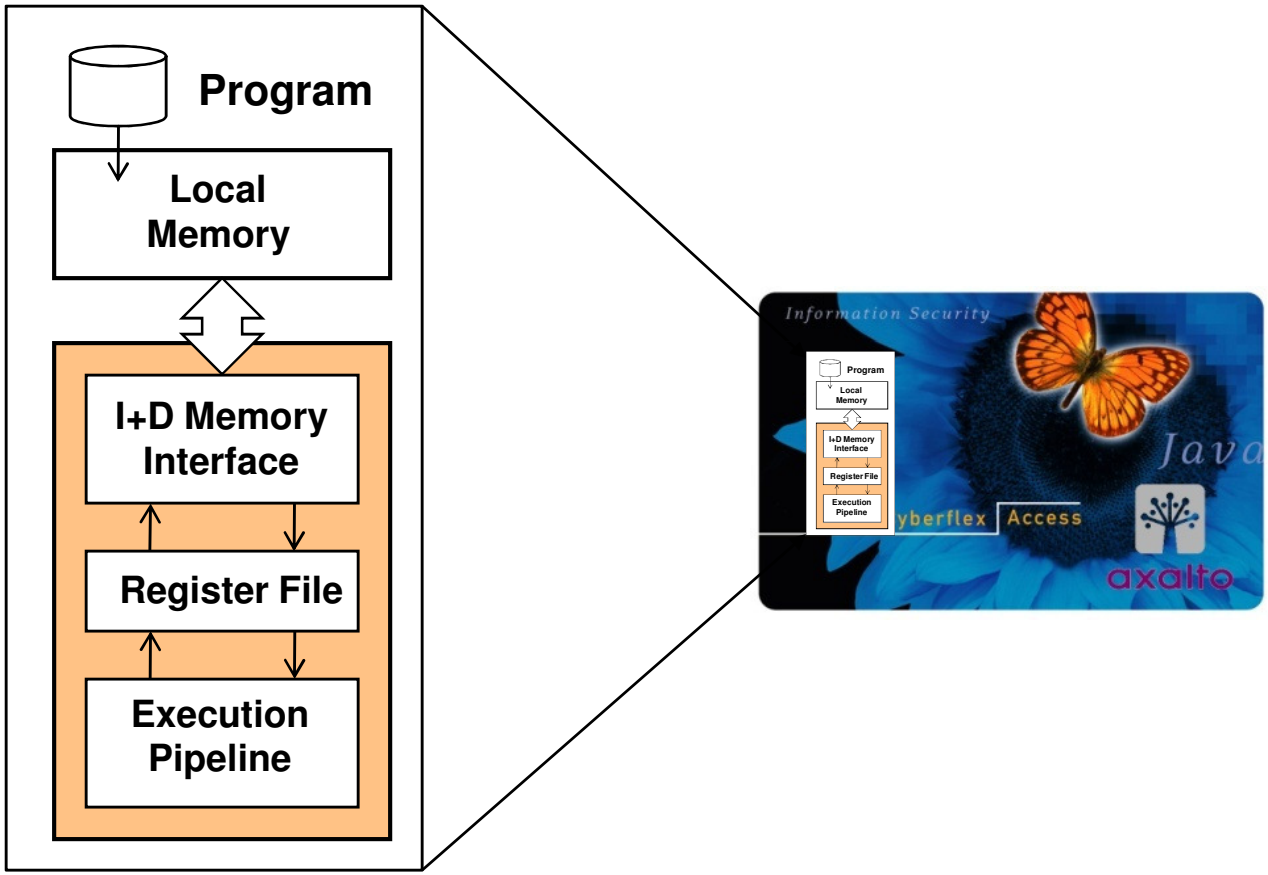
- **How to implement trustworthy secure embedded systems**
  - **that can thwart attacks?**
  - **that are efficient?**



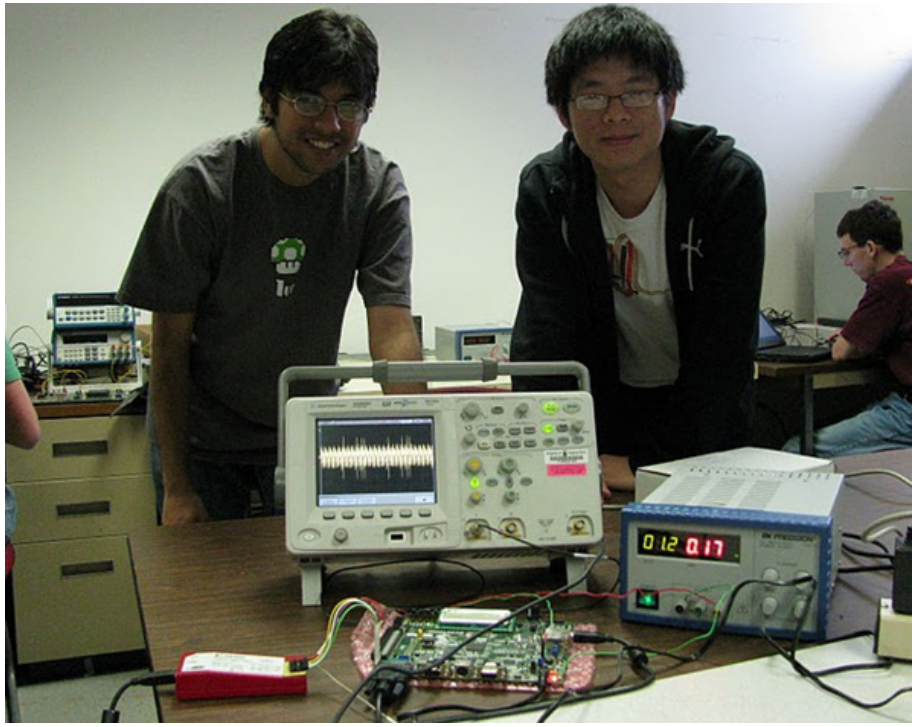
Two examples of ongoing projects

1. **Preventing Implementation Attacks on Software**
2. **Chip-Unique Binding of Software and Hardware**

Starting Point: An Embedded Core



Passive Attack



Side-channel Analysis:

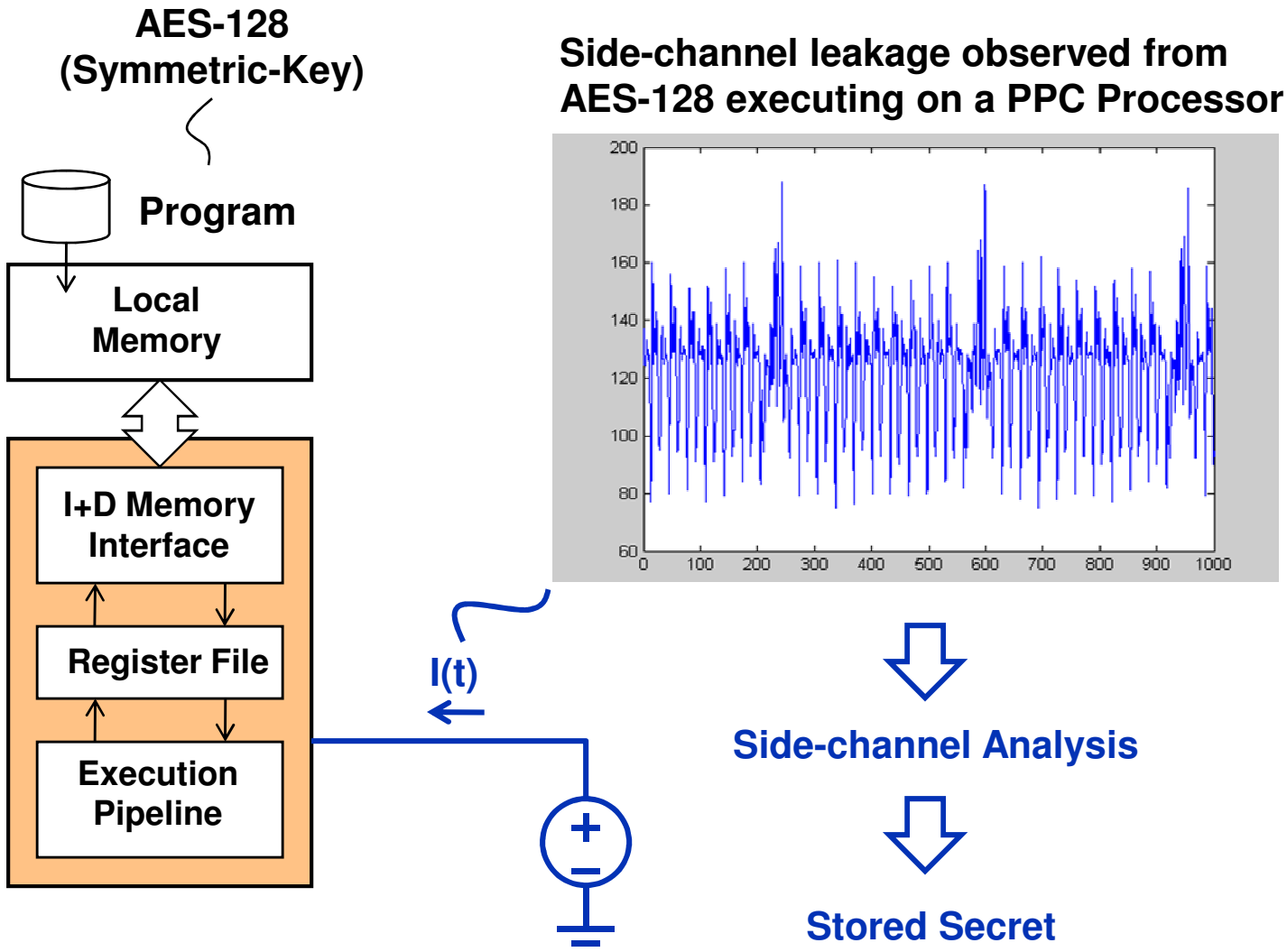
AES-128 (symmetric-key) on a embedded 32-bit CPU

- 256 measurements ("traces") disclose first key byte

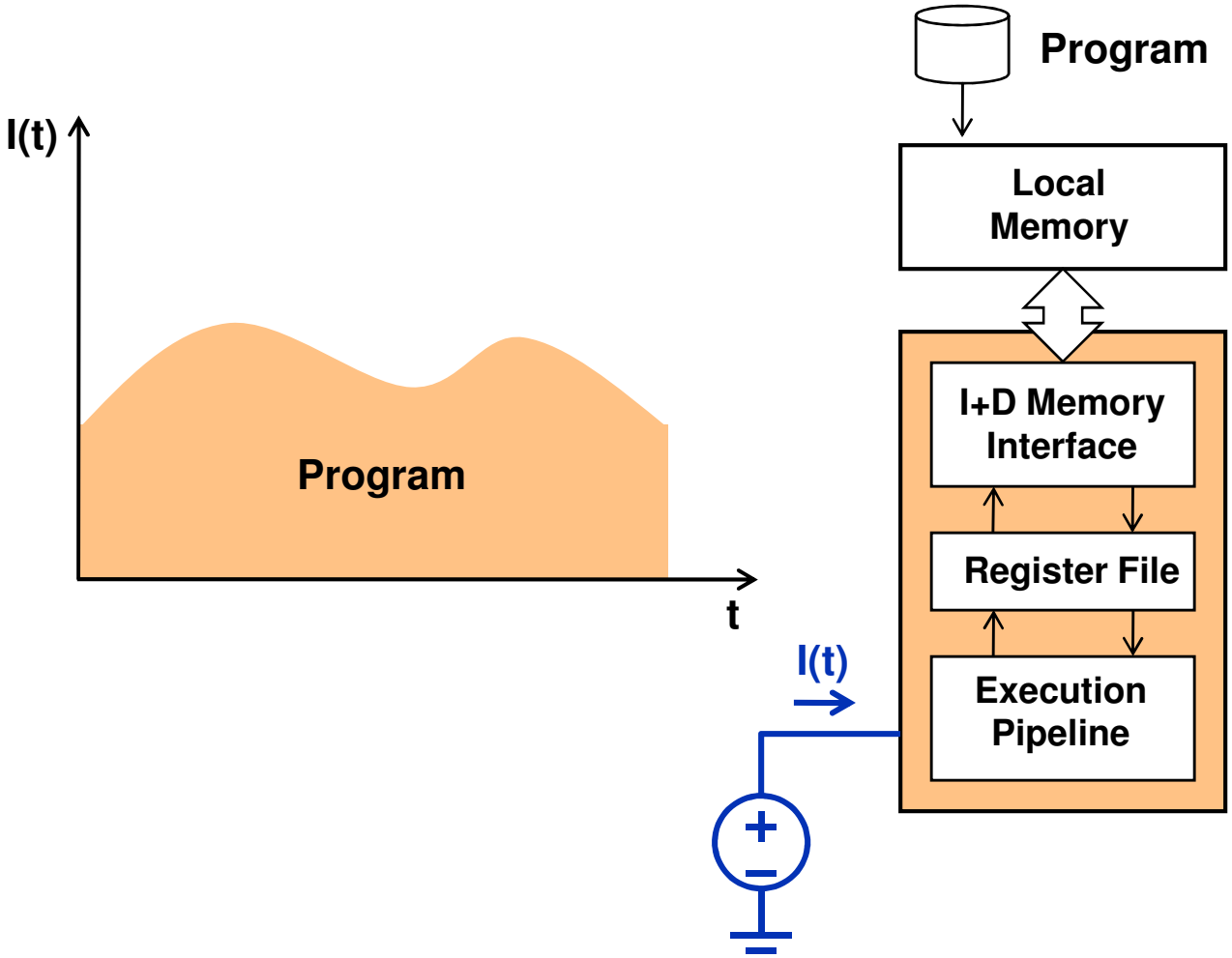
- 40,960 traces disclose ALL key bytes

Real-time for attack ~ 5 minutes

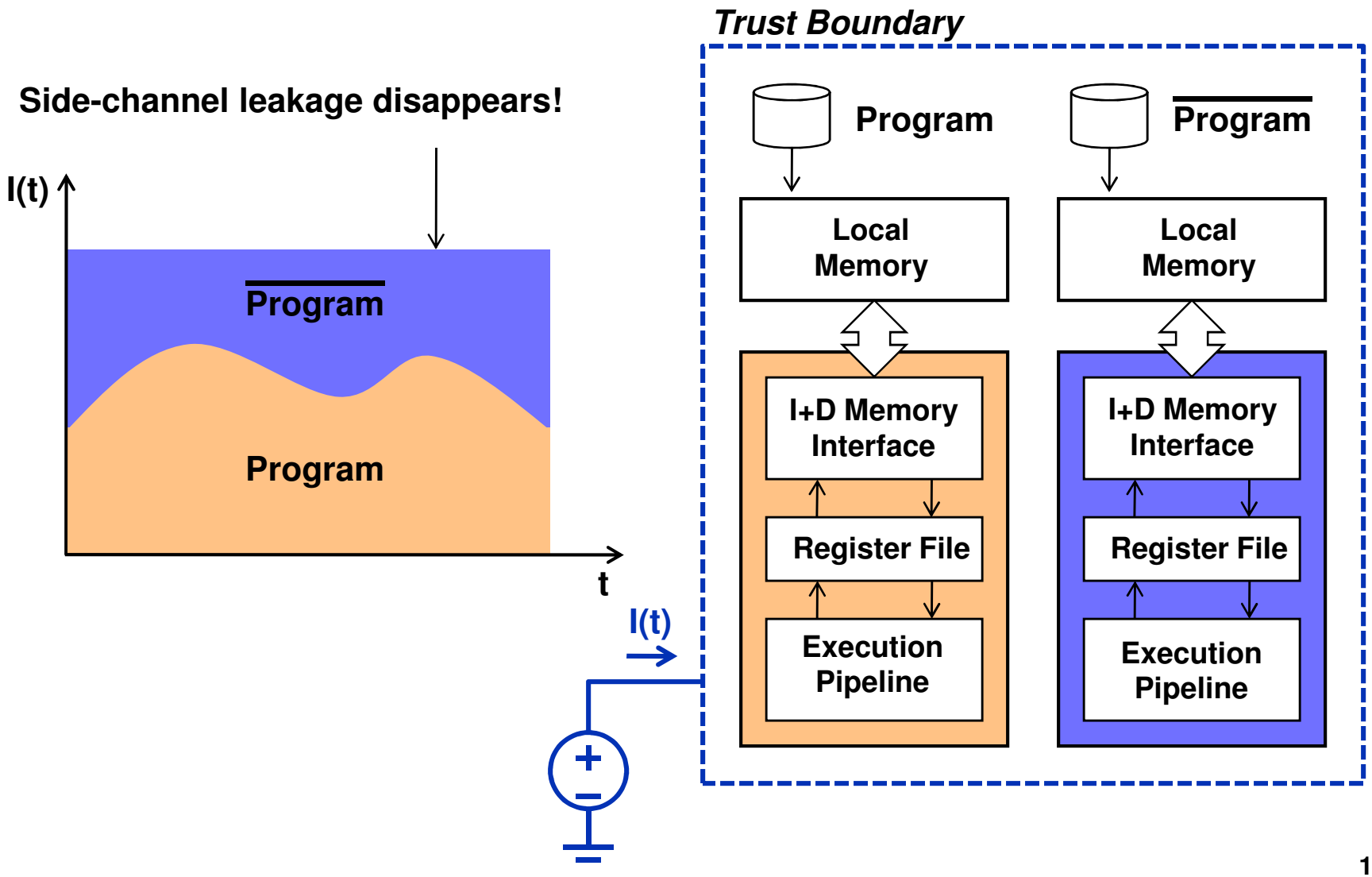
Implementation Attack



How to thwart implementation attack?



How to thwart implementation attack?



How to write Program and $\overline{\text{Program}}$?

- **Side Channel Leakage is proportional to the Hamming Weight of the Sensitive Data**

How to write Program and $\overline{\text{Program}}$?

- **Side Channel Leakage is proportional to the Hamming Weight of the Sensitive Data**

- **Program and $\overline{\text{Program}}$ work on *complementary* sensitive data**

  If Program writes `0x55` into register `r5`
  then $\overline{\text{Program}}$ writes `0xAA` into register `r5`

- **Program and $\overline{\text{Program}}$ execute *complementary* instructions**

  If Program performs `and r3, r4, r5`
  the $\overline{\text{Program}}$ performs `or r3, r4, r5`

- **Program and $\overline{\text{Program}}$ run synchronized**

How to write Program and $\overline{\text{Program}}$?

- **Side Channel Leakage is proportional to the Hamming Weight of the Sensitive Data**

- **Program and $\overline{\text{Program}}$ work on *complementary* sensitive data**

  If Program writes `0x55` into register `r5`
  then $\overline{\text{Program}}$ writes `0xAA` into register `r5`

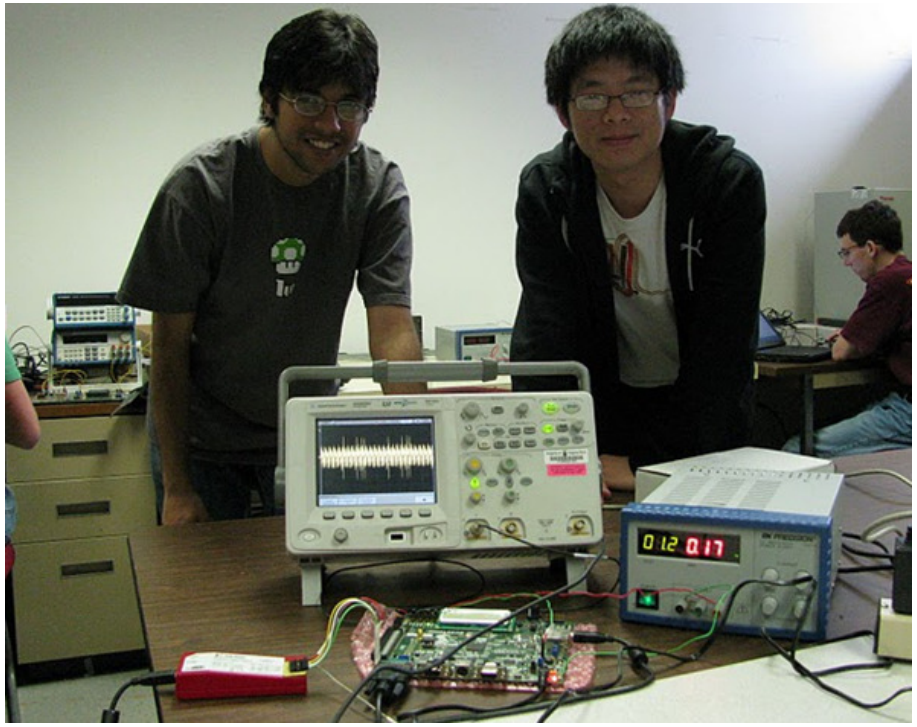
- **Program and $\overline{\text{Program}}$ execute *complementary* instructions**

  If Program performs `and r3, r4, r5`
  the $\overline{\text{Program}}$ performs `or r3, r4, r5`

- **Program and $\overline{\text{Program}}$ run synchronized**

⇨ **Hamming Weight of Sensitive Data remain constant**

Resulting Side-channel strength



Side-channel Analysis:

AES-128 (symmetric-key) on a dual-core CPU with complementary programs

- **81920** traces to disclose first key byte (single-core: 256 traces)

- **1M** traces cannot disclose all key bytes (single-core: 40960 traces discloses all)

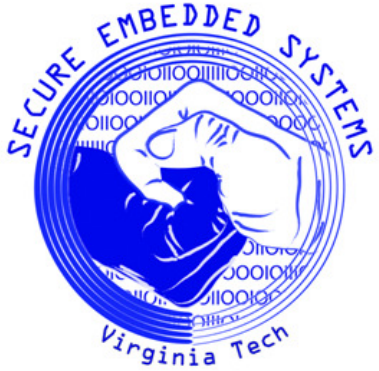
Of course, there are other attacks ...

- **Invasive attacks breach the trust boundary; Non-invasive attacks do not**
- **Active attacks affect the trusted behavior; Passive attacks do not**

	Active	Passive
Invasive	Tampering	Probing
Non-Invasive	Fault Attack	Side-channel Attack (SCA)

Our Research

- **How to implement trustworthy secure embedded systems**
  - **that can thwart attacks?**
  - **that are efficient?**

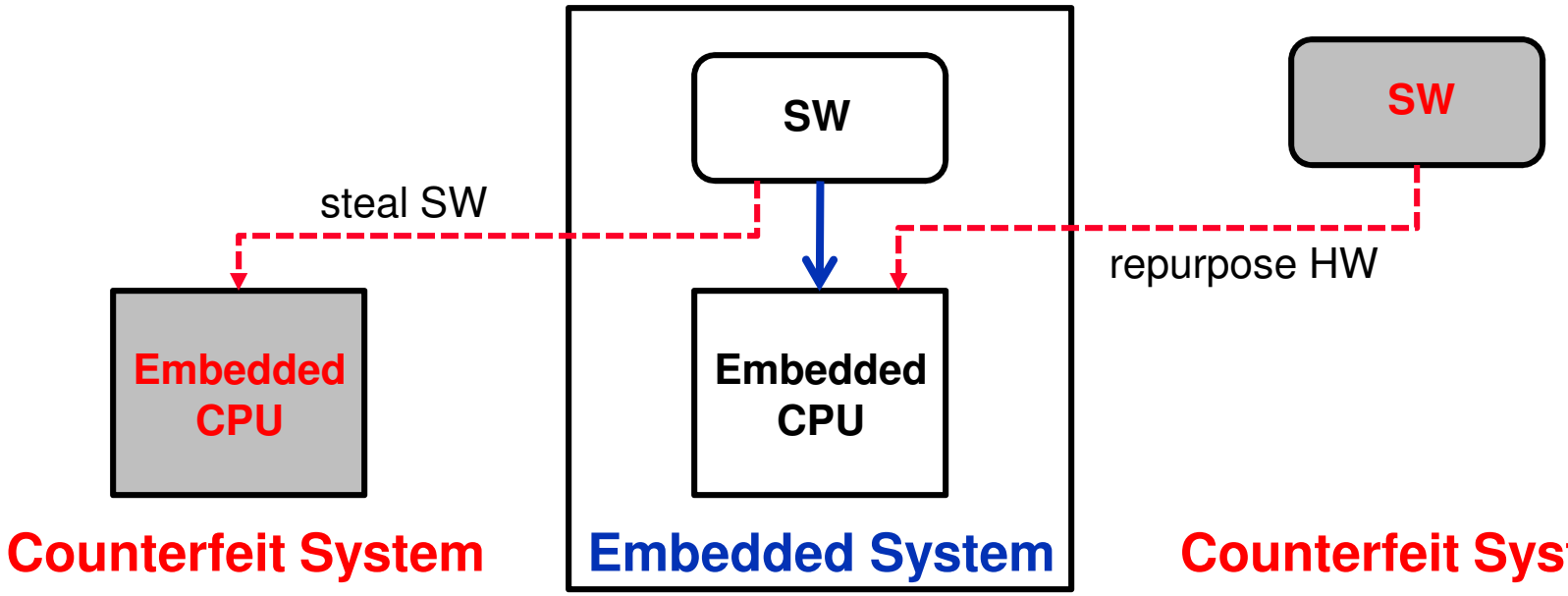


<u>Two examples of ongoing projects</u>

1. **Preventing Implementation Attacks on Software**
2. **Chip-Unique Binding of Software and Hardware**

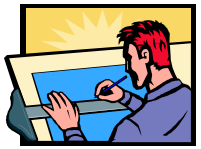
Chip-Unique Binding of SW and HW

- **How can we demonstrate the uniqueness of the link between embedded hardware and embedded software ?**



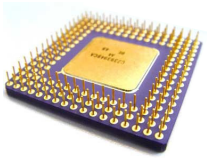
Physical Unclonable Functions

Designer



Chip Fab

Chip



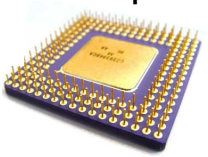
An unclonable on-chip ID is a chip-level structure that deliberately exploits random process manufacturing variations to establish the chip identity

Random Process Manufacturing Variations

- **Wire width**
- **Doping Level**
- **Threshold Voltage**

Chip1

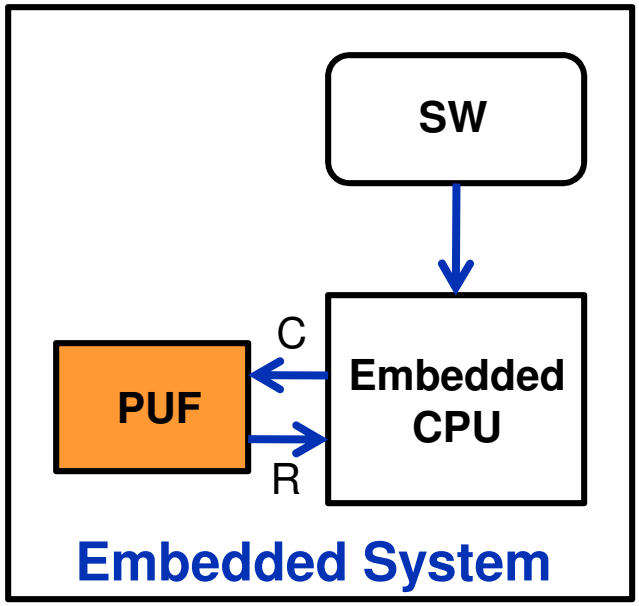
Chip3



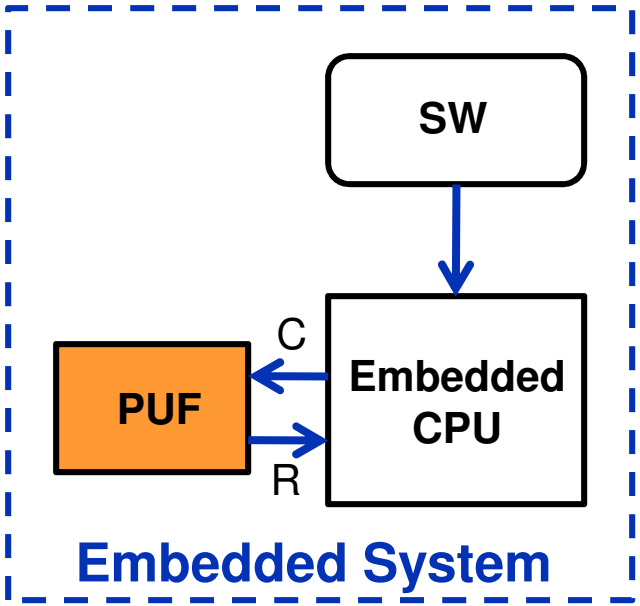
Chip2

Chip-Unique Binding of SW and HW

- **By definition, a PUF cannot be copied or tampered with**
- **A PUF can be implemented as a challenge/response function**
- **A PUF works can be used as an *intrinsic* key generator**



SW Binding with a PUF



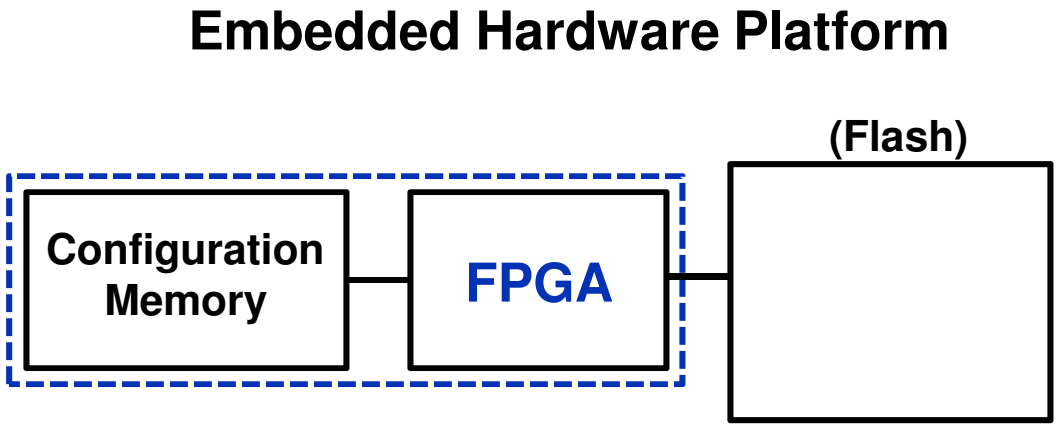
1. **PUF Enrollment**

   Generate a C/R pair
   Encrypt Software
   $E_R(SW)$
   Distribute
   C, $E_R(SW)$

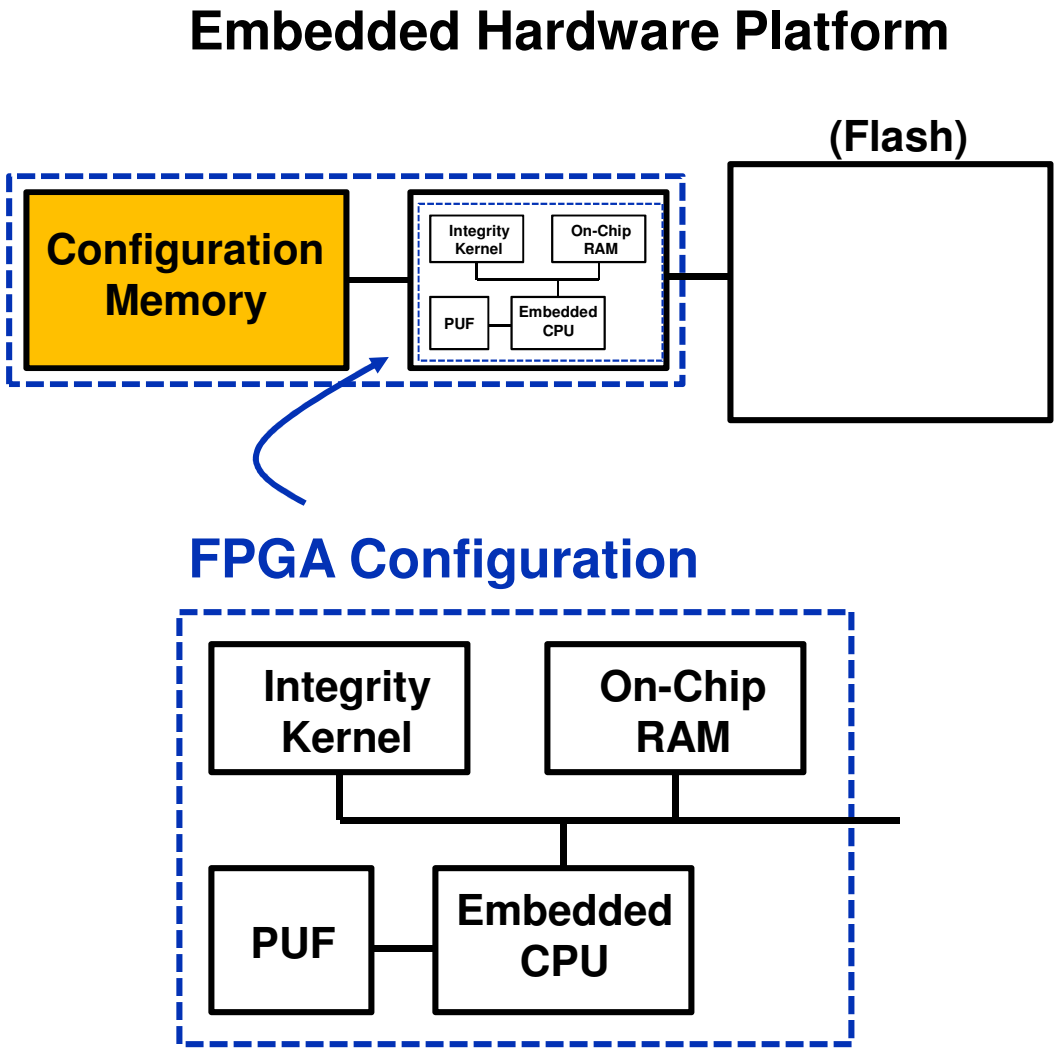
2. **Deployment**

   Recreate R with C
   Decrypt Software
   $D_{PUF(C)}(SW)$
   Execute SW

Protection FPGA SW and HW



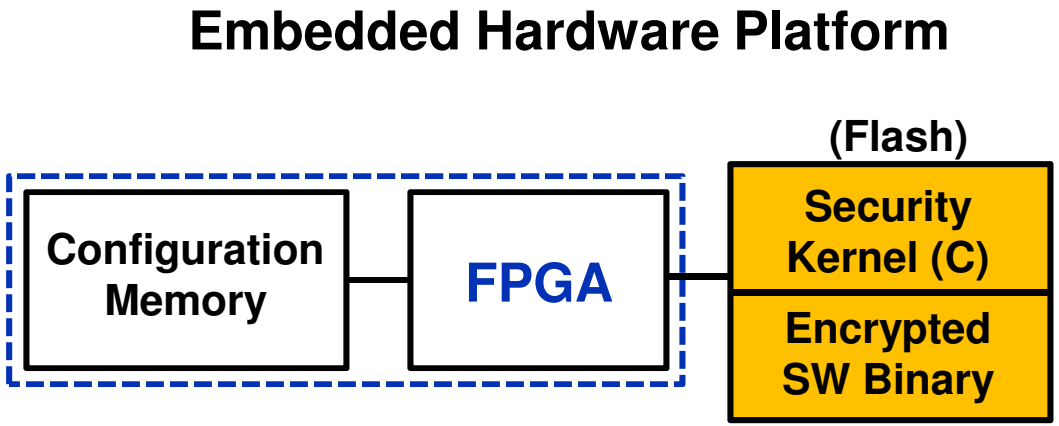
Protection FPGA SW and HW

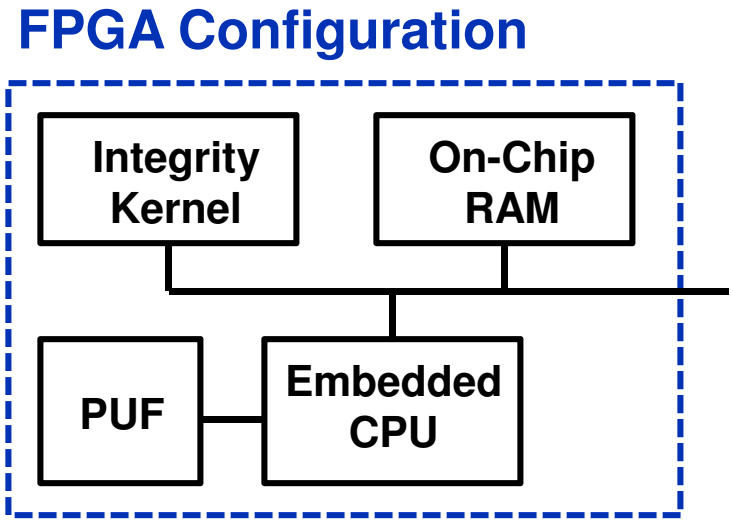


1. **Configure FPGA**

   Define HW

Protection FPGA SW and HW





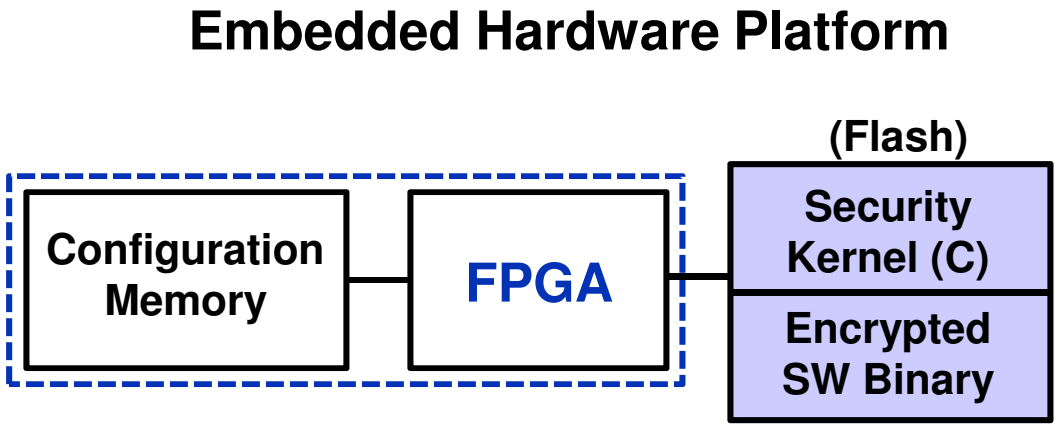
1. **Configure FPGA**

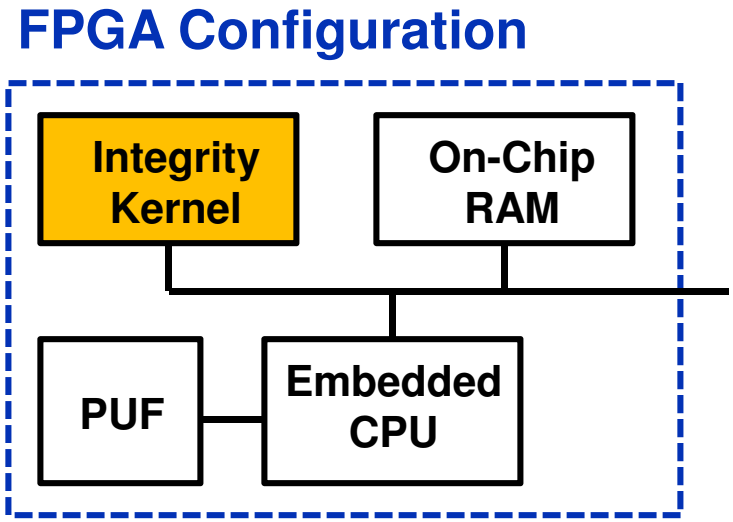
   Define HW

2. **Prepare SW**

   Encrypt SW w/ PUF R
   Store PUF C

Protection FPGA SW and HW





1. **Configure FPGA**

   Define HW

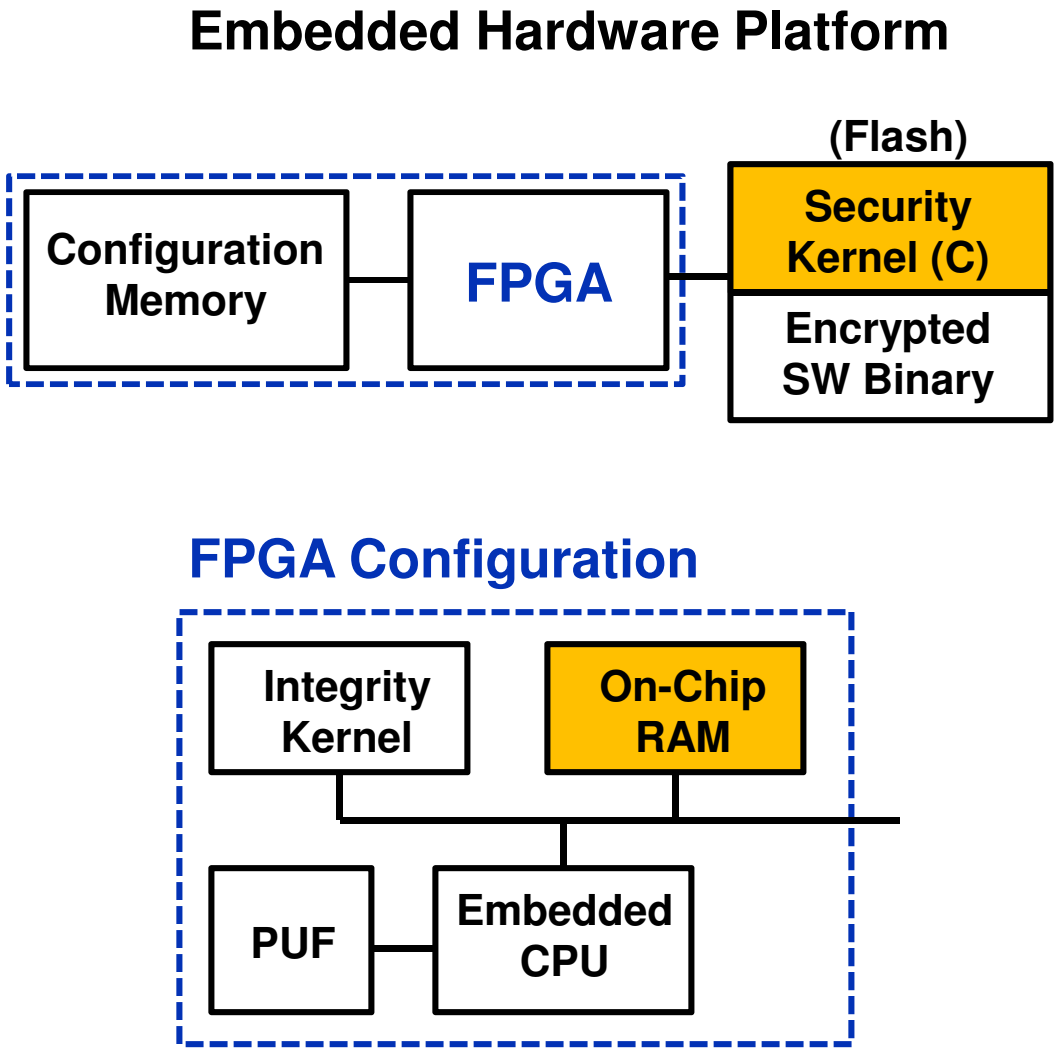
2. **Prepare SW**

   Encrypt SW w/ PUF R
   Store PUF C

3. **Boot System**

   Verify Flash Integrity

Protection FPGA SW and HW



1. **Configure FPGA**

   Define HW

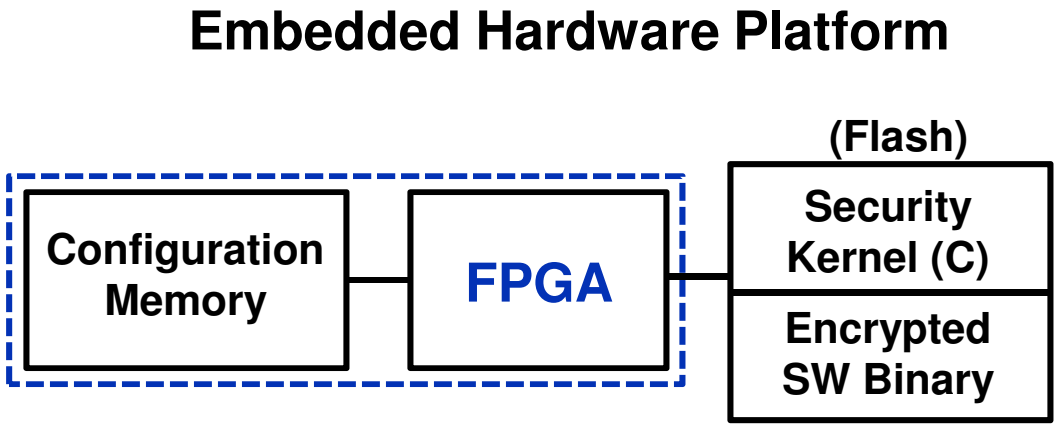
2. **Prepare SW**

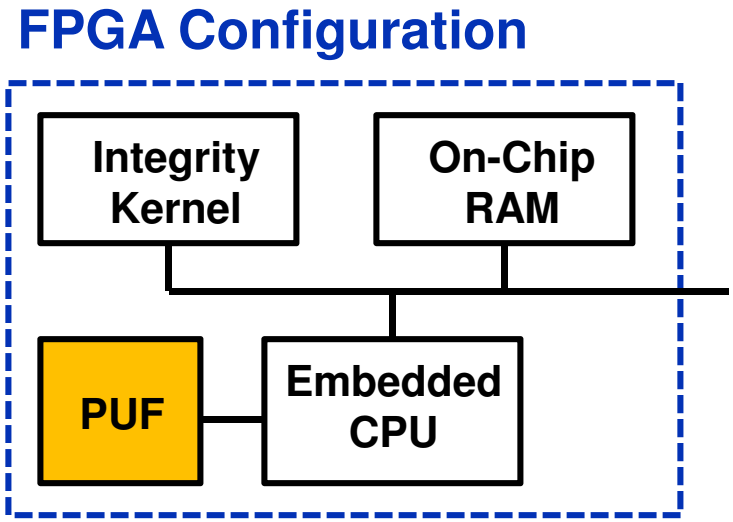
   Encrypt SW w/ PUF R
   Store PUF C

3. **Boot System**

   Verify Flash Integrity
   Load Security Kernel

Protection FPGA SW and HW





1. **Configure FPGA**

   Define HW

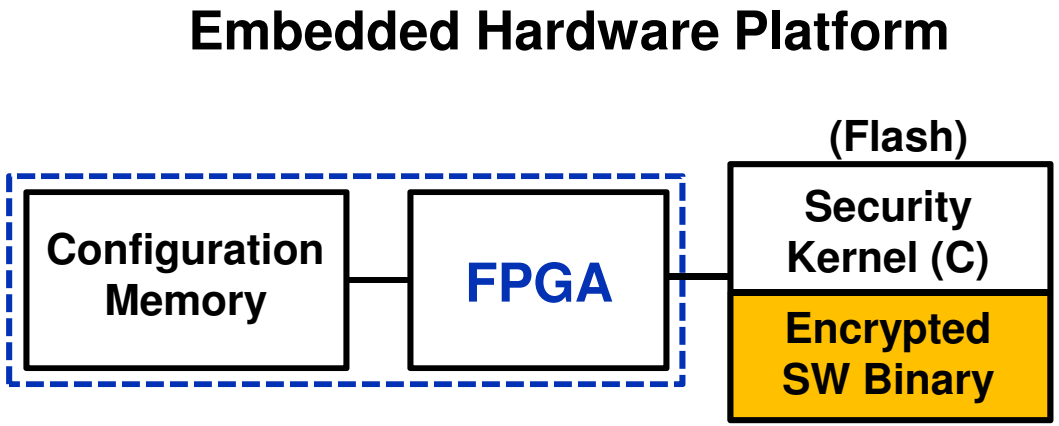
2. **Prepare SW**

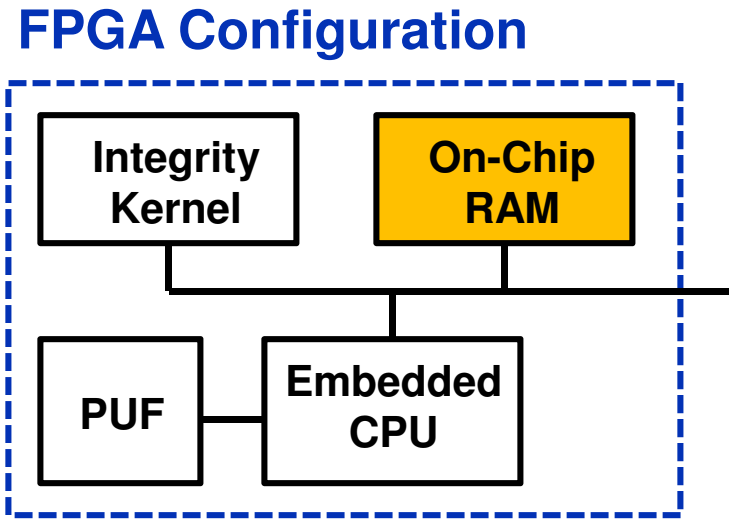
   Encrypt SW w/ PUF R
   Store PUF C

3. **Boot System**

   Verify Flash Integrity
   Load Security Kernel
   Retrieve Response
   Load & Decrypt SW

Protection FPGA SW and HW





1. **Configure FPGA**

   Define HW

2. **Prepare SW**

   Encrypt SW w/ PUF R
   Store PUF C

3. **Boot System**

   Verify Flash Integrity
   Load Security Kernel
   Retrieve Response
   Load & Decrypt SW
   Execute!

Conclusion

- **Secure Embedded Systems =**
  Information Security +
  Efficient Implementation +
  Trustworthy Implementation

- **The Hardware/Software Symbiosys:**
  Software delivers complexity, flexibility
  Hardware delivers trustworthiness