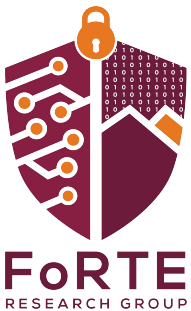
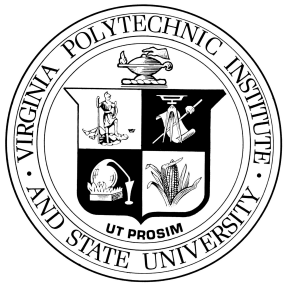
RIPencapsulation: Defeating IP Encapsulation on TI MSP Devices

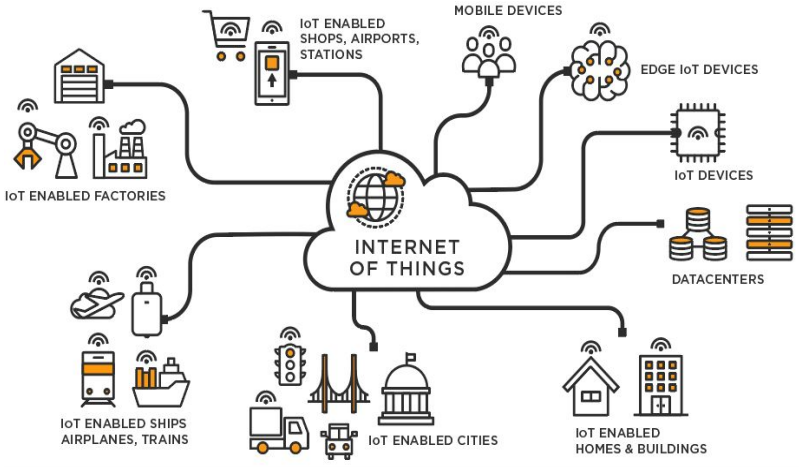
Prakhar Sah and Matthew Hicks

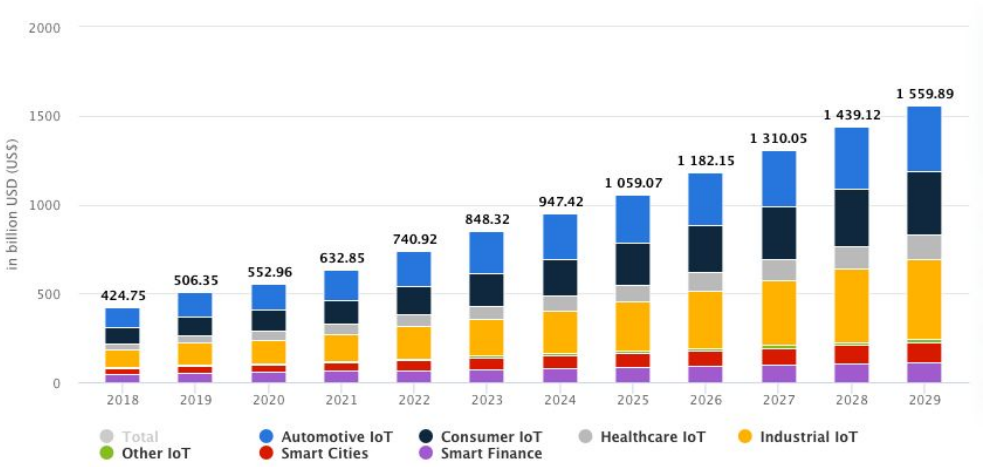




18th USENIX WOOT Conference on Offensive Technologies – August 12-13, 2024

The Future is IoT



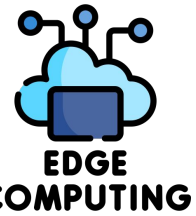


[1] Statista. 2020. Internet of Things (IoT Statistics Report).

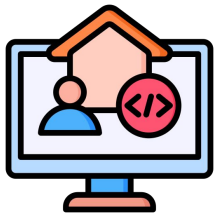
IoT – Security Landscape

IoT devices to house 79 ZB of data by 2025[1]

Intellectual Property & Proprietary ML Algorithms



Physical access is the common case

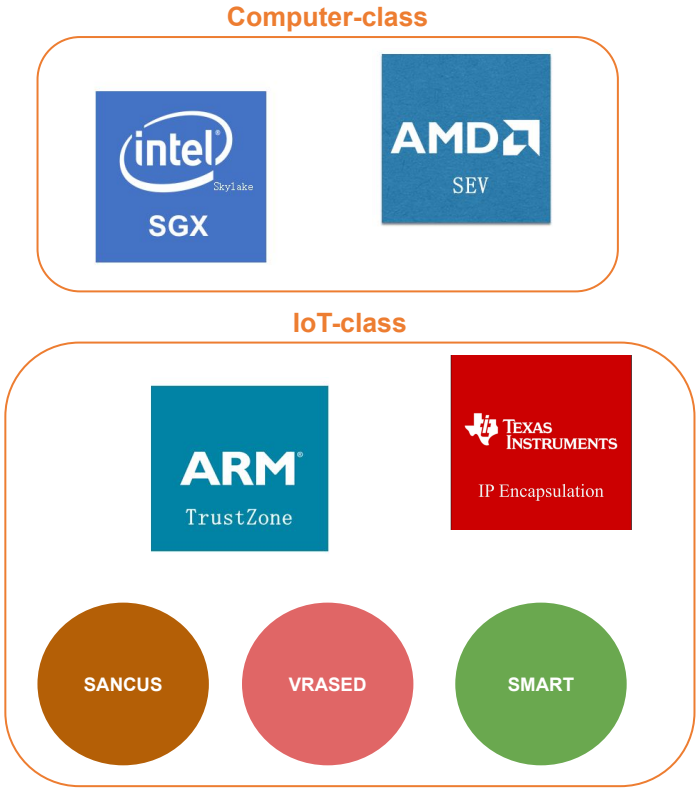


Co-resident software threat

[1] Statista. 2020. Internet of Things (IoT Statistics Report).

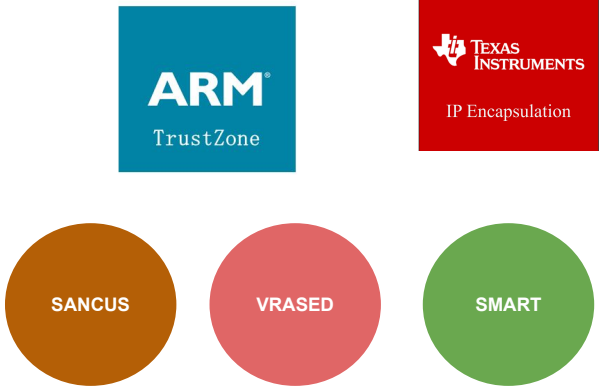
What is all the fuss about TEEs?

- Trusted Execution Environments address the threat of co-resident software
- Bifurcate hardware into high-security and low-security domains
- Range of off-the-shelf TEEs to choose from
- Computer-class vs IoT-class TEEs



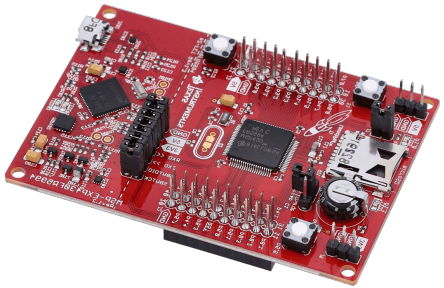
TEEs in the context of IoT Devices

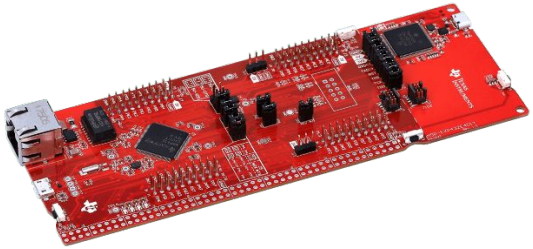
- No fancy features like hardware memory-based isolation, pointer authentication, full-fledged hypervisors …
- Energy and hardware area constraints dictate lightweight designs
- Generally suited for preventing IP theft of library-based proprietary algorithms



Texas Instruments MSP Microcontroller Series

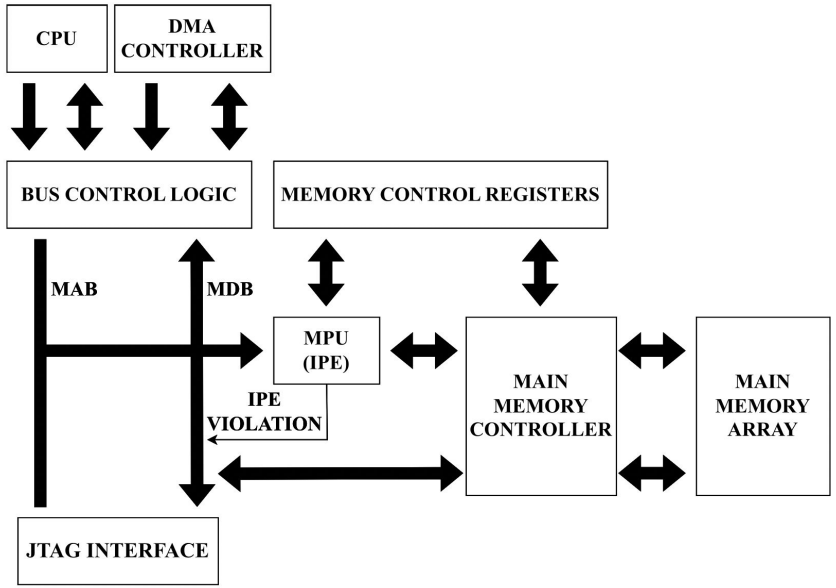
- Low-power industrial grade microcontrollers
- MSP430 => 16-bit instruction set
- MSP432 => 32-bit instruction set
- Widely deployed in ultra-low power applications and wireless networks
- Feature regularly in IoT research





IP Encapsulation – The TI MSP TEE

- As name suggests, meant for protecting (proprietary) code and data
- MPU (MSP430) or SYSCTL (MSP432) snoop on memory address bus and PC
- Only program code executed from inside IPE region has data access privileges



Two flaws in IPE design

We discovered two vulnerabilities in TI MSP IPE design –

Vulnerability 1: When execution leaves the IPE zone unexpectedly, the contents of the CPU register file remain

Implication: *If IPE exits can be forced non-destructively, possibility of reconstructing IPE firmware using intermediate register states*

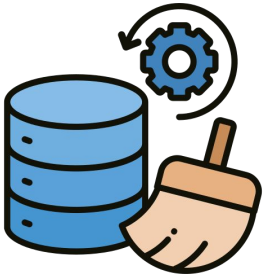
Vulnerability 2: IPE allows all code outside IPE zone to branch to arbitrary instructions within the IPE zone

Implication: *Use arbitrary branch capability to reuse IPE code (gadgets) and bypass read/write access control checks*

Building blocks – Unsanitized Context Switches

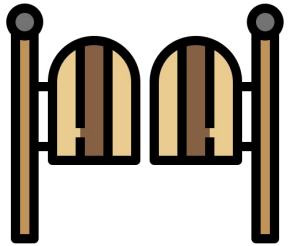
Transition between high and low-security domains

Intermediate execution state like CPU registers



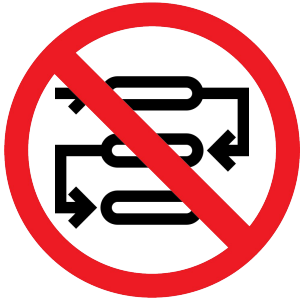
TEE should sanitize execution state on context switches

Building blocks – No Call Site Verification



Untrusted code interacts with TEE at “call sites”

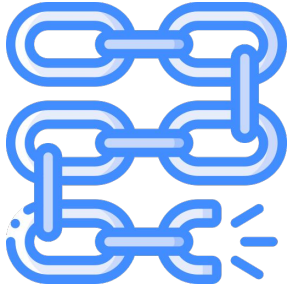
Call site verification checks execute access to callee



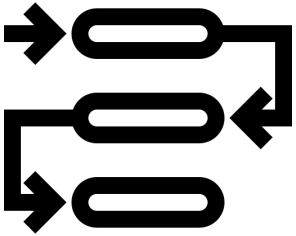
Prevents program control flow manipulation and execution of arbitrary commands

Background – Code-reuse Attacks

When injecting malicious code is prevented



Chain existing code snippets to create “gadgets”



Manipulate program control flow to execute arbitrary commands

Not quite there yet…

1. Cannot set breakpoints
2. Need a way to exit and re-enter IPE in a incrementing loop
3. Debugger HALT signals do not provide single-instruction granularity
4. What if general interrupts are disabled on IPE entry?
5. How can gadgets be identified without knowledge of the firmware binary?

Not quite there yet…

1. Cannot set breakpoints
2. Need a way to exit and re-enter IPE in a incrementing loop
3. Debugger HALT signals do not provide single-instruction granularity
4. What if general interrupts are disabled on IPE entry?
5. How can gadgets be identified without knowledge of the firmware binary?

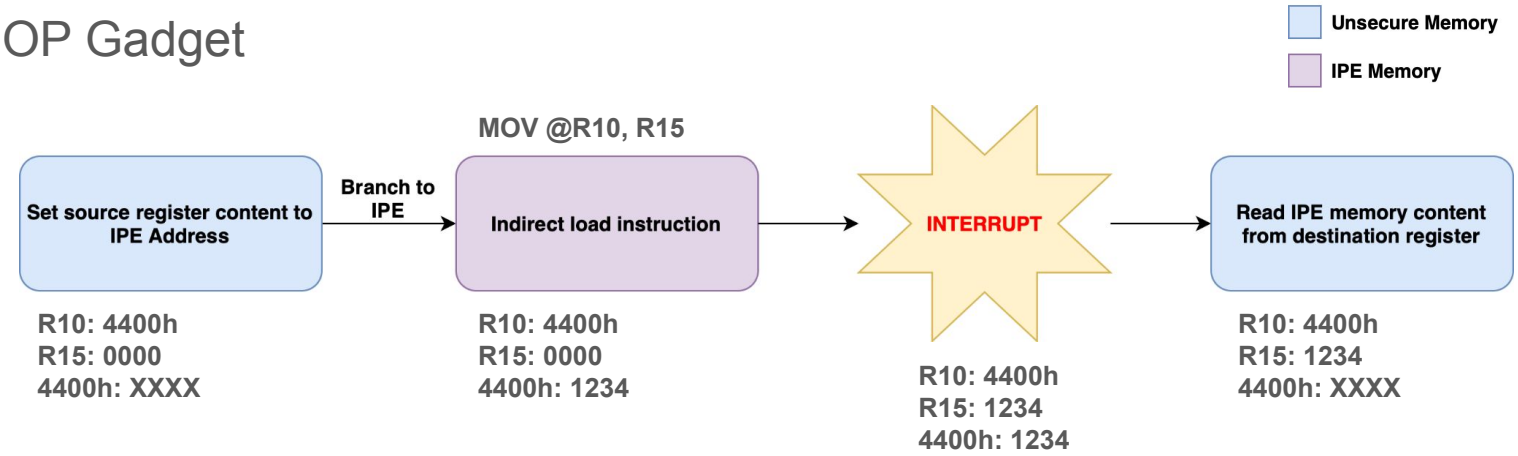
Interrupt Oriented Programming

- Taking inspiration from return oriented programming and replacing `ret` with carefully crafted interrupt
- Use timer interrupts to interrupt program at single-cycle granularity (sub-instruction level)
- IPE cannot protect fixed memory section containing Interrupt Vector Table[2] – gives ability to handle interrupts maliciously in unsecure memory

[2] Texas Instruments. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx family user's guide.

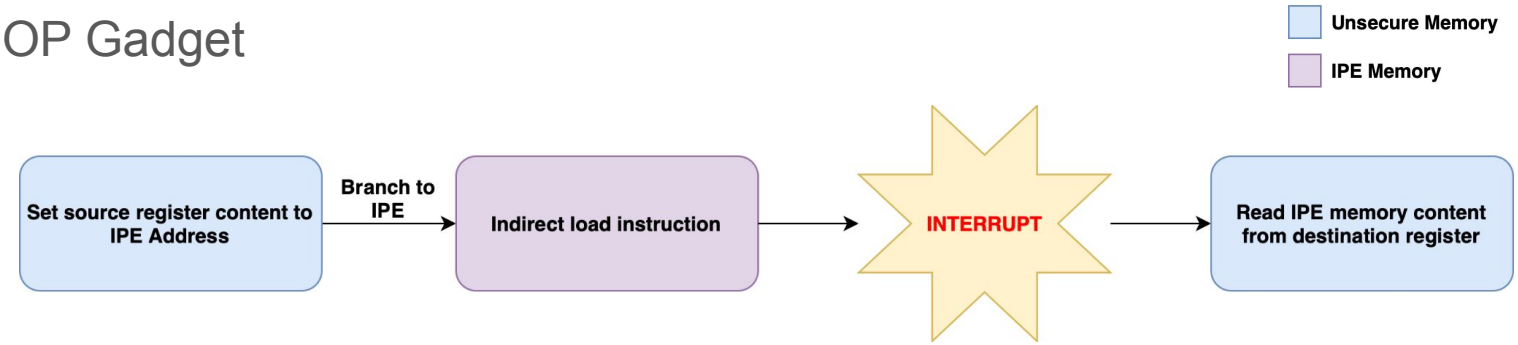
Interrupt Oriented Programming

Read IOP Gadget

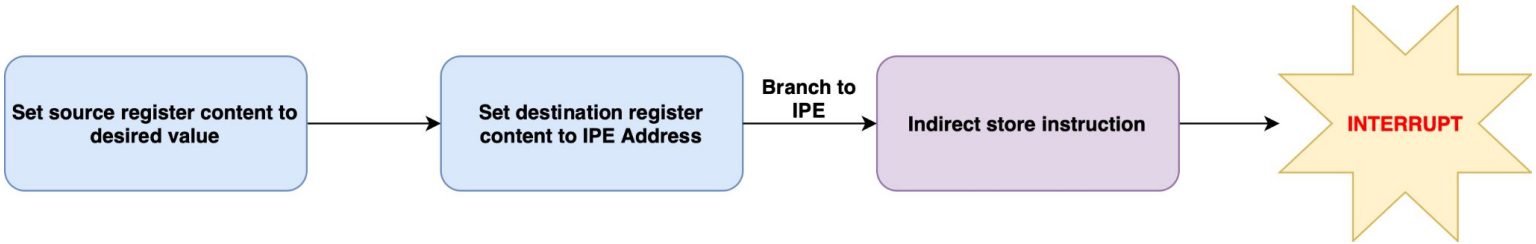


Interrupt Oriented Programming

Read IOP Gadget



Write IOP Gadget



Not quite there yet…

1. Cannot set breakpoints
2. Need a way to exit and re-enter IPE in a incrementing loop
3. Debugger HALT signals do not provide single-instruction granularity
4. What if general interrupts are disabled on IPE entry?
5. How can gadgets be identified without knowledge of the firmware binary?

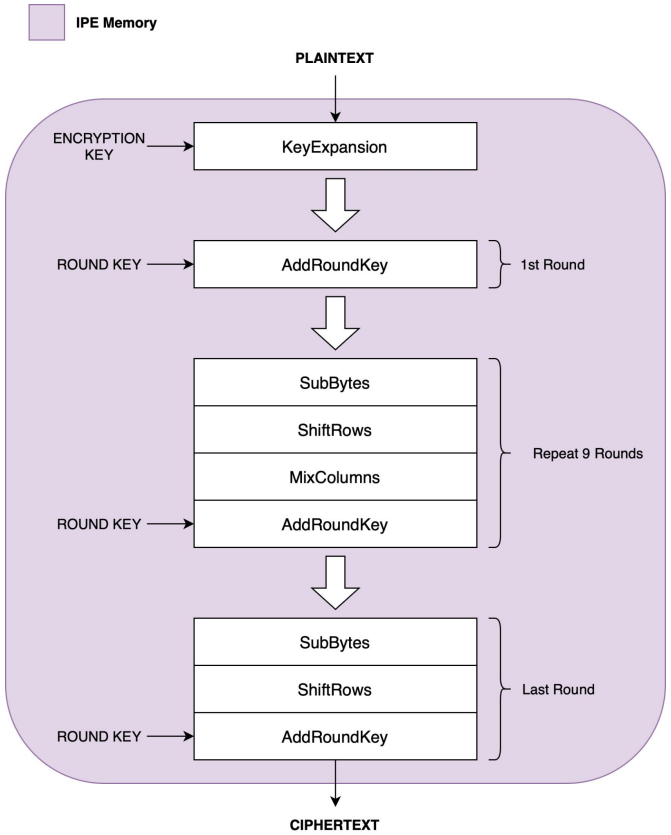
Introducing RIPencapsulation

"**RIPencapsulation** combines **interrupt-based** control flow attack patterns, with **data oriented** attack patterns, and **side-channel** attack patterns to gain **unrestricted read, write and execute access** to IPE-protected memory"

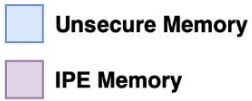
- Use Vulnerability 1 to identify IOP gadgets without a priori knowledge of IPE binary
- Vulnerability 2 allows skipping general interrupt disabling instruction during IPE call

Example IPE Firmware: tinyAES Encryption

- Lightweight 10 rounds AES-128 ECB implementation
- Store entire encryption implementation (including 128-bit secret key) inside IPE zone as library function
- Unsecure code passes the 128-bit plaintext to tinyAES IPE function
- tinyAES returns 128-bit ciphertext back to the unsecure code
- tinyAES code and secret key is a blackbox



Phase 1 – Creating a Side Channel



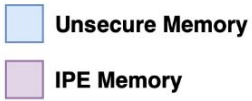

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



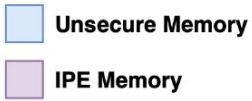

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



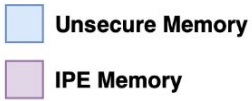

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



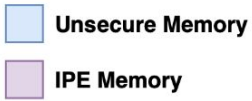

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



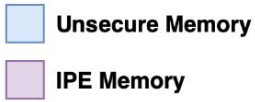

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

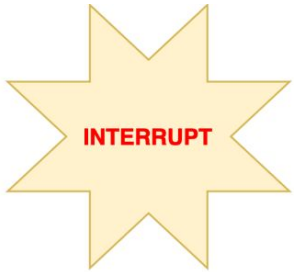
```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel





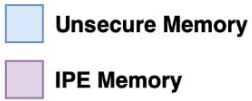

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



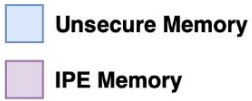

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



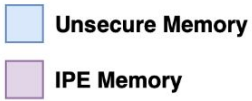

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



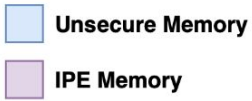

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



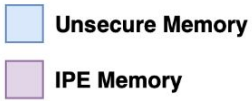

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



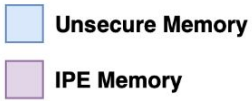

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



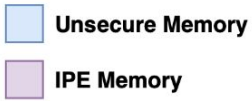

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



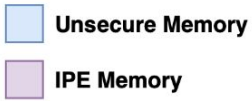

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel



Malicious Code():

```
1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

tinyAES():

```
1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

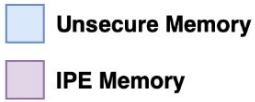
KeyExpansion():

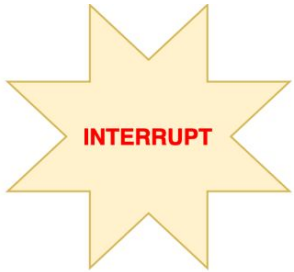
```
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

Malicious ISR():

```
1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel





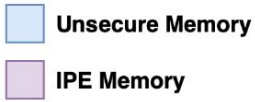

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

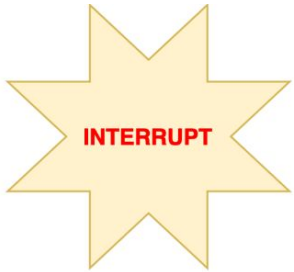
```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel





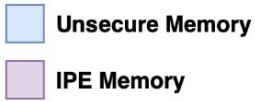

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

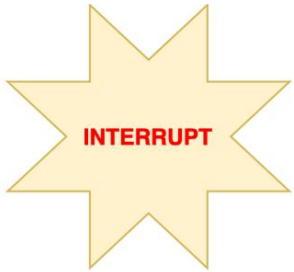
```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel





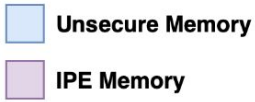

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

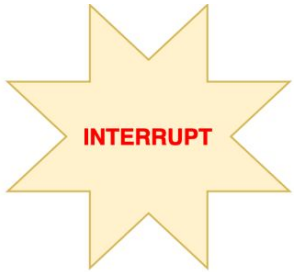
```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel





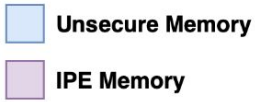

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

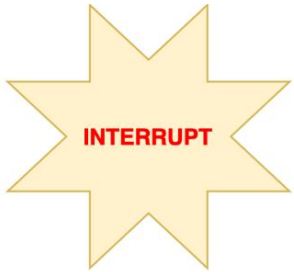
```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel





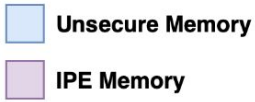

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

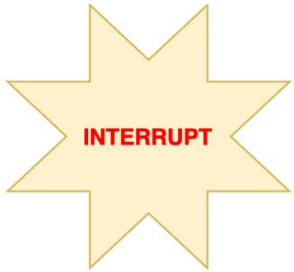
```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 1 – Creating a Side Channel






```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 2 – IPE State Dump Post-processing

- Easy to identify register indirect load instructions like `MOV @Rn, Rm` in register dumps

- Set `Rn`: Unsecure Address =>

  Jump to `MOV @Rn, Rm` =>

  Interrupt IPE process =>

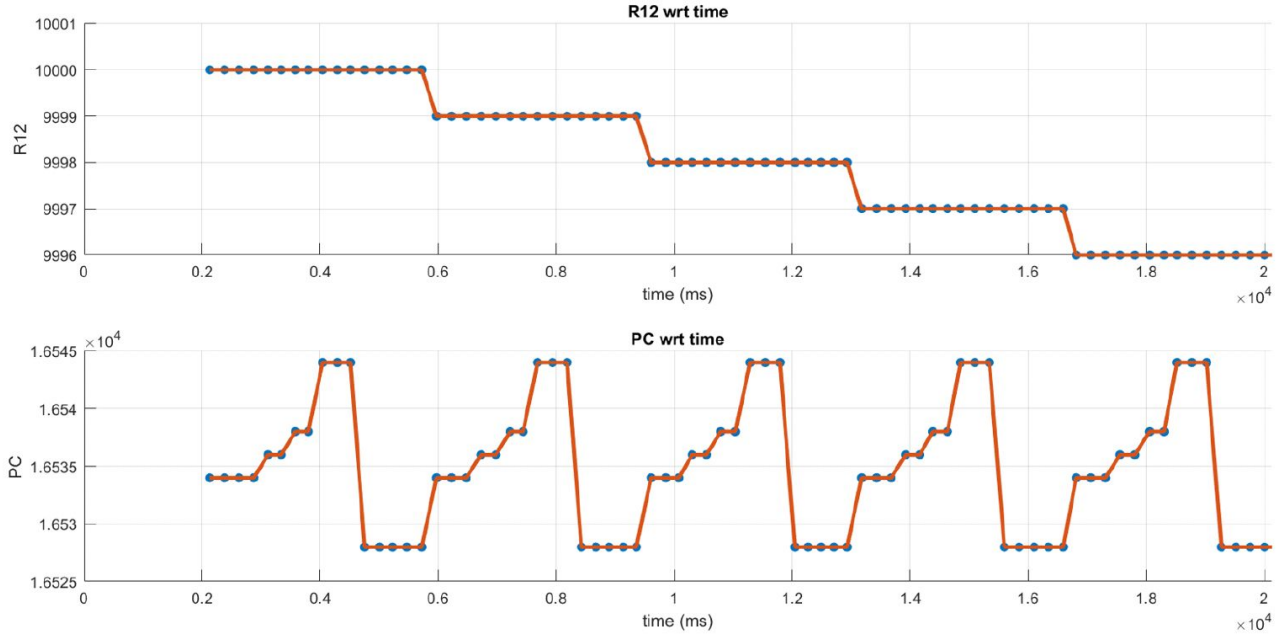
  Read `Rm` from register dumps

- Almost all register direct and indirect instructions can be identified

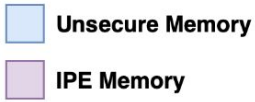
Source Operand	Destination Operand				
	Rm	PC	x(Rm)	TONI	&TONI
Rn	✓	✓	✓	×	×
@Rn	✓	✓	✓	×	×
@Rn+	✓	✓	✓	×	×
#N	-	-	-	×	×
x(Rn)	✓	✓	✓	×	×
EDE	×	×	×	×	×
&EDE	×	×	×	×	×

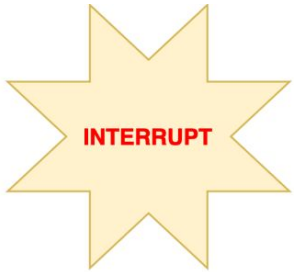
Phase 2 – IPE State Dump Post-processing

- Extract more information using cycle count for each instruction
- Recurring PC patterns imply loops



Phase 2 – IPE State Dump Post-processing





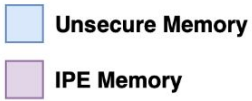

```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 2 – IPE State Dump Post-processing



Malicious Code():

```
1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

tinyAES():

```
1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

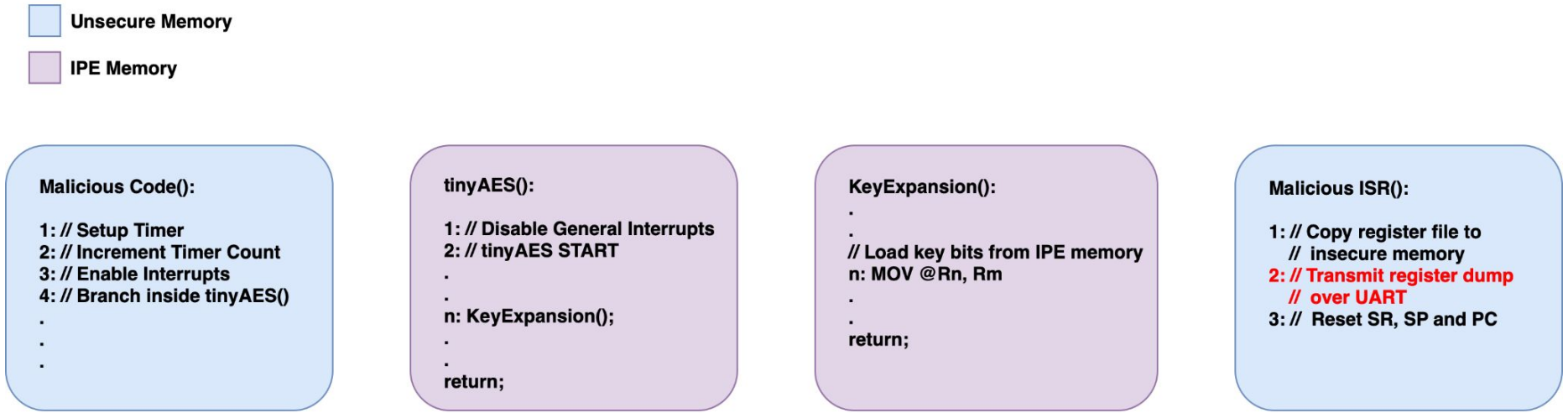
KeyExpansion():

```
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

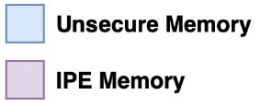
Malicious ISR():

```
1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 2 – IPE State Dump Post-processing



Phase 2 – IPE State Dump Post-processing




```
Malicious Code():

1: // Setup Timer
2: // Increment Timer Count
3: // Enable Interrupts
4: // Branch inside tinyAES()
.
.
.
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets

- Indirect load instructions like `MOV @Rn, Rm` can also be used as IOP gadgets for read exploits

- Set `Rn`: IPE Address =>

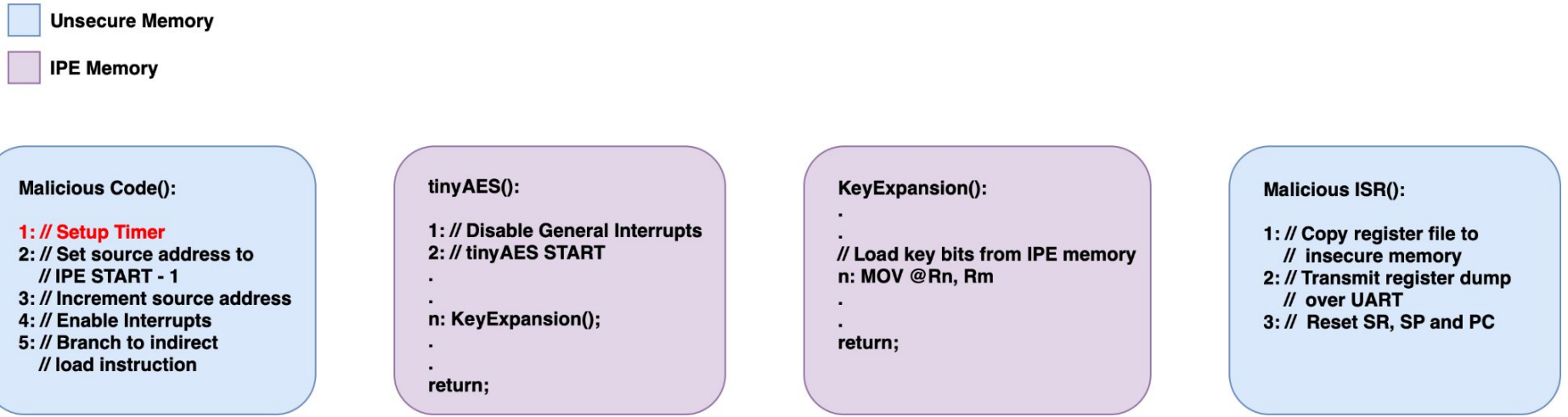
  Jump to `MOV @Rn, Rm` =>

  Interrupt IPE process =>

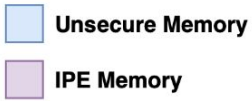
  Read `Rm` from register dumps

- Exfiltrate entire IPE firmware

Phase 3 – IPE Memory Access Gadgets



Phase 3 – IPE Memory Access Gadgets



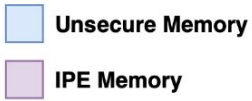

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



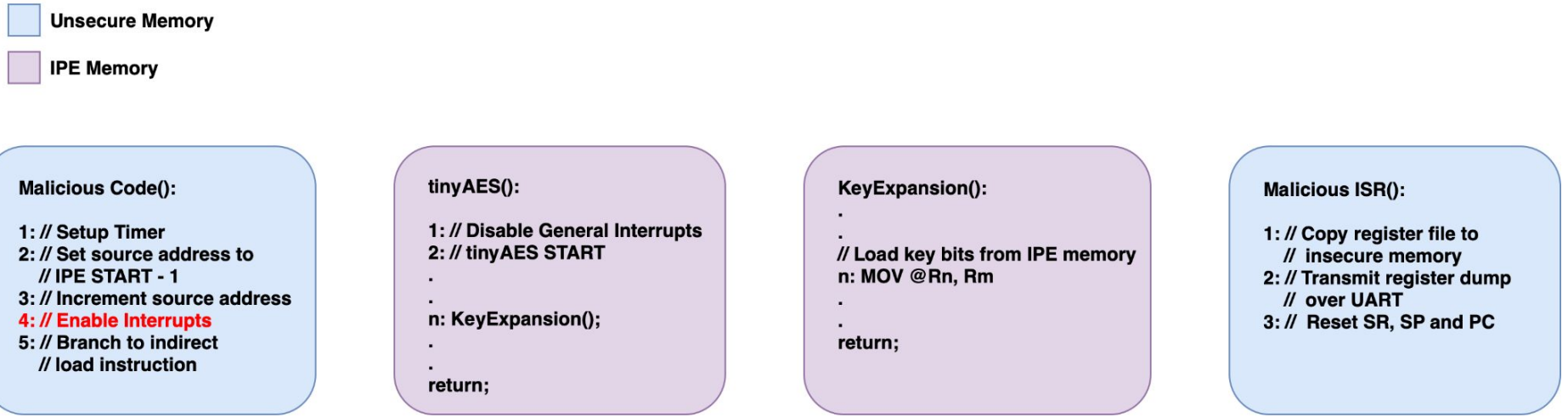

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

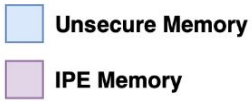
```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



Phase 3 – IPE Memory Access Gadgets



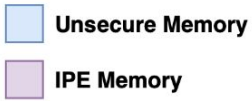

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



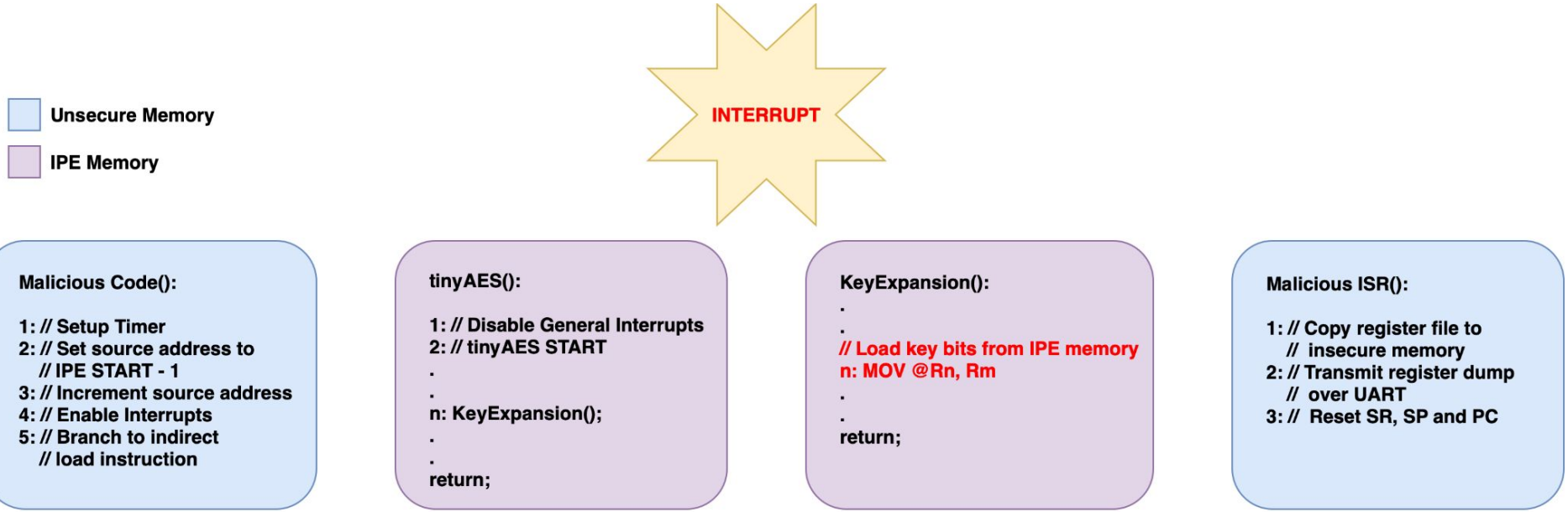

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

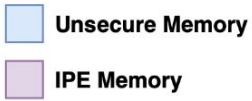
```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



Phase 3 – IPE Memory Access Gadgets



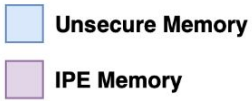

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



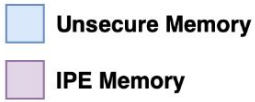

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



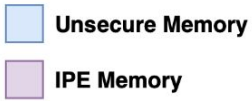

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



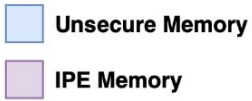

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



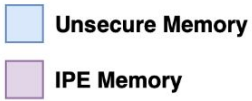

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



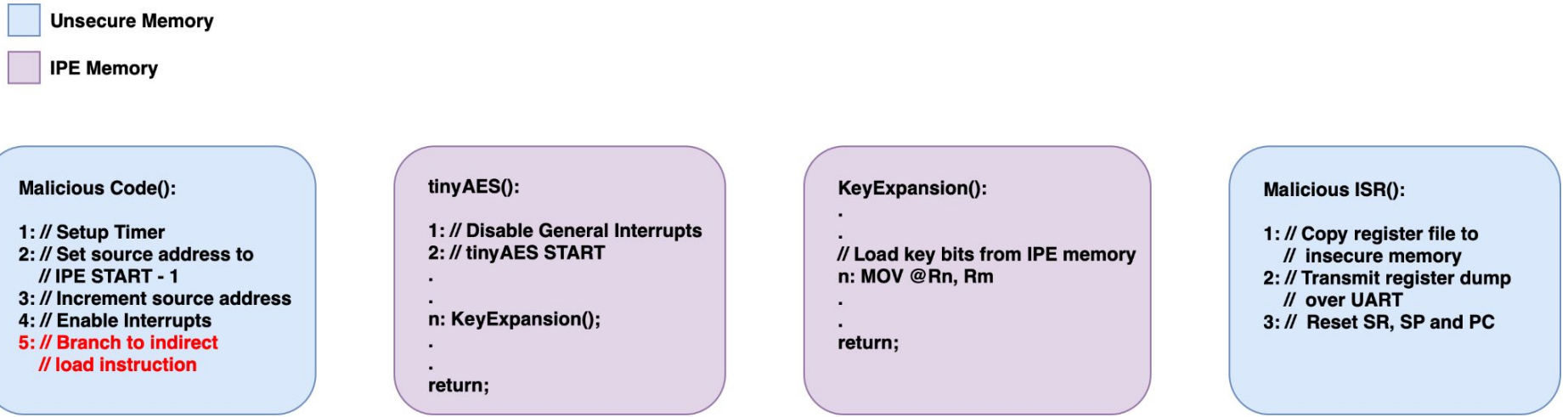

```
Malicious Code():

1: // Setup Timer
2: // Set source address to
   // IPE START - 1
3: // Increment source address
4: // Enable Interrupts
5: // Branch to indirect
   // load instruction
```

```
tinyAES():

1: // Disable General Interrupts
2: // tinyAES START
.
.
n: KeyExpansion();
.
.
return;
```

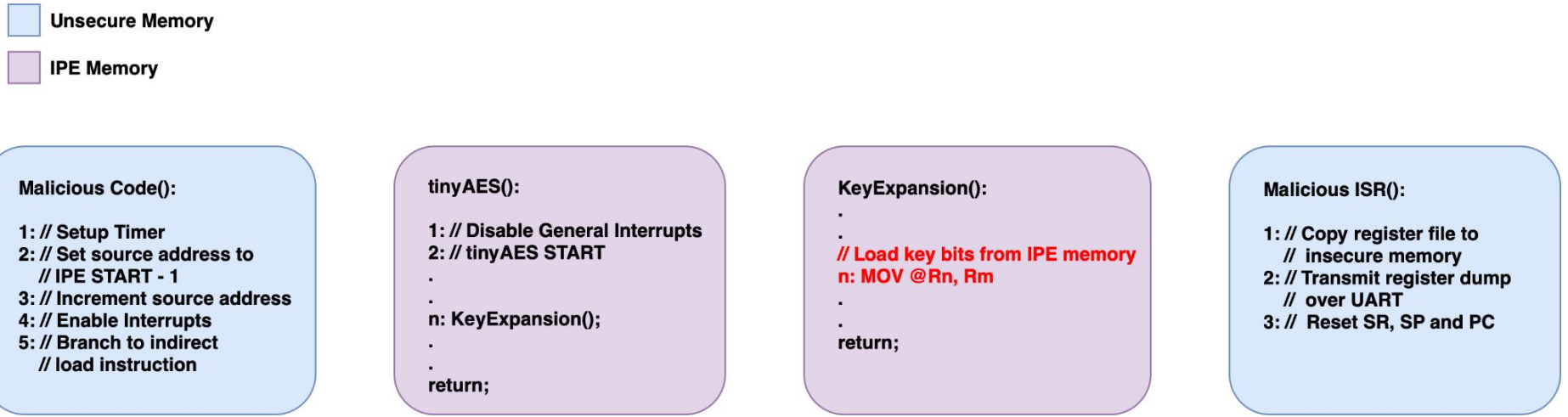
```
KeyExpansion():
.
.
// Load key bits from IPE memory
n: MOV @Rn, Rm
.
.
return;
```

```
Malicious ISR():

1: // Copy register file to
   //  insecure memory
2: // Transmit register dump
   //  over UART
3: //  Reset SR, SP and PC
```

Phase 3 – IPE Memory Access Gadgets



Phase 3 – IPE Memory Access Gadgets



Write Exploit

- Indirect store instructions like `MOV Rn, x(Rm)`

- Same procedure as read exploits

Hence, RIPencapsulation breaks all guarantees of **Confidentiality and Integrity**

RIPencapsulation on MSP432

- Different Instruction Set
- Interrupt signal stays asserted leading to immediate re-interrupt

```
Malicious ISR():

1: // Stop timer
2: // Clear interrupt flags
3: // Copy register file to
   //  insecure memory
4: // Transmit register dump
   //  over UART
5: // Write malicious return
   //  address to MSP
6: return;
```

RIPencapsulation on MSP432

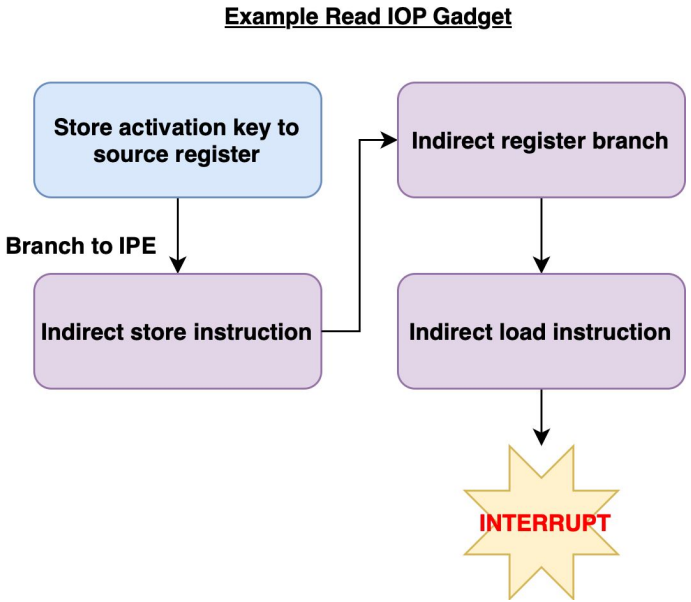
- Different Instruction Set
- Interrupt signal stays asserted leading to immediate re-interrupt
- Interrupt remains active if proper return not followed

Malicious ISR():

```
1: // Stop timer
2: // Clear interrupt flags
3: // Copy register file to
   //  insecure memory
4: // Transmit register dump
   //  over UART
5: // Write malicious return
   //  address to MSP
6: return;
```

RIPencapsulation on MSP432

- Different Instruction Set
- Interrupt signal stays asserted leading to immediate re-interrupt
- Interrupt remains active if proper return not followed
- Data access must be unlocked for each IPE access from unsecure world



Testbench Setup

- Setup IP Encapsulation according to TI-recommended secure programming practices
- Disable interrupts at the start and clear the register file at the end of the IPE code
- Tests conducted on 16-bit and 32-bit MSP devices
- Source code compiled using msp430-gcc
- Source code and detailed instructions can be found at RIPencapsulation's git repository[3]

Platform	MSP430FR5994	MSP432P401R
ISA	MSP430 CPUX	ARMv7-M
NV Memory Type	FRAM	Flash
NV Memory Size	256 KB	256 KB
Max Clock Frequency	16 MHz	48 MHz
IP Encapsulation	Yes	Yes
UART Interface	Yes	Yes
Timer Module	Yes	Yes

[3] https://github.com/FoRTE-Research/RIPencapsulation

All compiled assemblies leak the secret in a matter of minutes!

AES-128 (TINY AES)

Optimization Level	Instructions decoded	Reveal key bits?	Contain Gadg
-O0	60.5%	✓	✓
-Og	66.6%	✓	✓
-O1	68.4%	✓	✓
-O2	68.4%	✓	✓
-O3	68.4%	✓	✓
-Os	67.9%	✓	✓
-Ofast	68.4%	✓	✓

< 120 sec

SADDI SHA

Optimization Level	Instructions decoded	Reveal 'pt' bits?	Cont Ga
-O0	59.4%	✓	✓
-Og	56.7%	✓	✓
-O1	53.9%	✓	✓
-O2	49.7%	✓	✓
-O3	50.1%	✓	✓
-Os	53%	✓	✓
-Ofast	50.1%	✓	✓

~ 5 min

GLADMAN SHA256

Optimization Level	Instructions decoded	Reveal 'pt' bits?	Contain Gadg
-O0	59.2%	✓	✓
-Og	60.9%	✓	✓
-O1	58.3%	✓	✓
-O2	60.4%	✓	✓
-O3	60.7%	✓	✓
-Os	57.7%	✓	✓
-Ofast	60.7%	✓	✓

~ 45 min

CODEBASE RSA

Optimization Level	Instructions decoded	Reveal key bits?	Cont Ga
-O0	58.3%	✓	✓
-Og	58.4%	✓	✓
-O1	57.3%	✓	✓
-O2	54.3%	✓	✓
-O3	54.3%	✓	✓
-Os	60.8%	✓*	✓
-Ofast	54.3%	✓	✓

< 5 min

* Readout from the stack is required for some bits of the private key

RIPencapsulation in the wild

- RIPencapsulation proves effective against a GPS/GPRS tracker application from a drone flight controller

-Opt	O0	Og	O1	O2	O3	Os	Ofast
IOP Gadget?	✓	✓	✓	✓	✓	✓	✓

- IP theft/imitation threat model, where a researcher/competitor (potential adversary) uses the licensed security critical library (e.g., flight controller) APIs in their development code

- Concurrent work by Bognar et al.[4] leverages RIPencapsulation primitives along with a call corruption vulnerability to carry out an end-to-end attack on MSP430 IPE

[4] Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers.

Why IP Encapsulation fails?

“The privacy of registers and on-chip caches should be protected by the trusted computing base from software attacks.”

- AEGIS by Edward Suh et al.

Call site verification at IPE entry and residual state clear up at IPE exit is a must!

Possible Patches

Secure Context Switches

- AEGIS-style secure context switching

Call Site Verification

- Use ARM-TrustZone-like gateway veneer combined with fault attack protection

Other Considerations

- Remember ultra-constrained nature of MSP devices
- Lightweight HW-SW co-design for adversarial model

Thank you for listening!