TZ-DATASHIELD: Automated Data Protection for Embedded Systems via Data-Flow-Based Compartmentalization

Zelun Kong[1], Minkyung Park[1], Le Guan[2], Ning Zhang[3], Chung Hwan Kim[1]

[1]University of Texas at Dallas

[2]University of Georgia

[3]Washington University in St. Louis

Data Security of MCU

Data Security of MCU

Microcontroller units are used in critical fields

Data Security of MCU

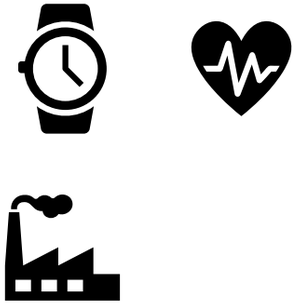
Microcontroller units are used in critical fields

- Healthcare

Data Security of MCU

Microcontroller units are used in critical fields

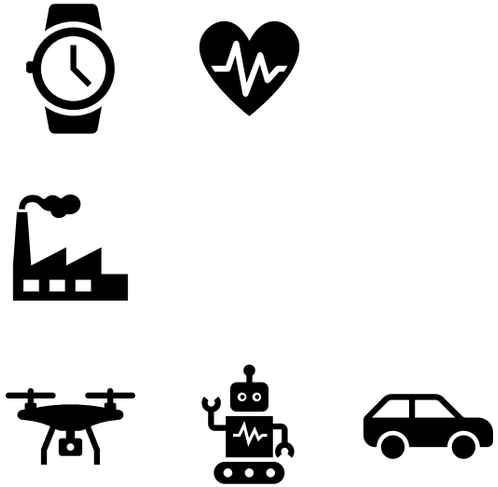
- Healthcare
- Industrial automation



Data Security of MCU

Microcontroller units are used in critical fields

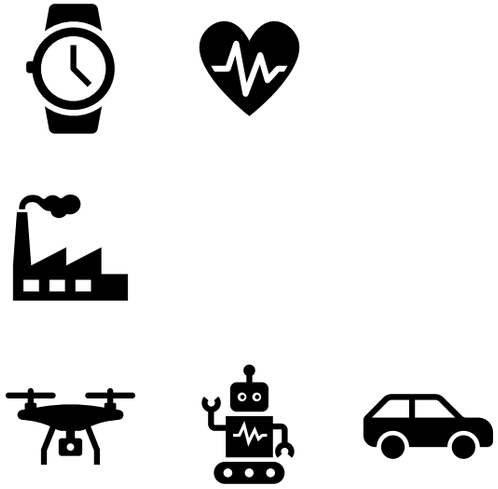
- Healthcare
- Industrial automation
- Autonomous driving vehicles

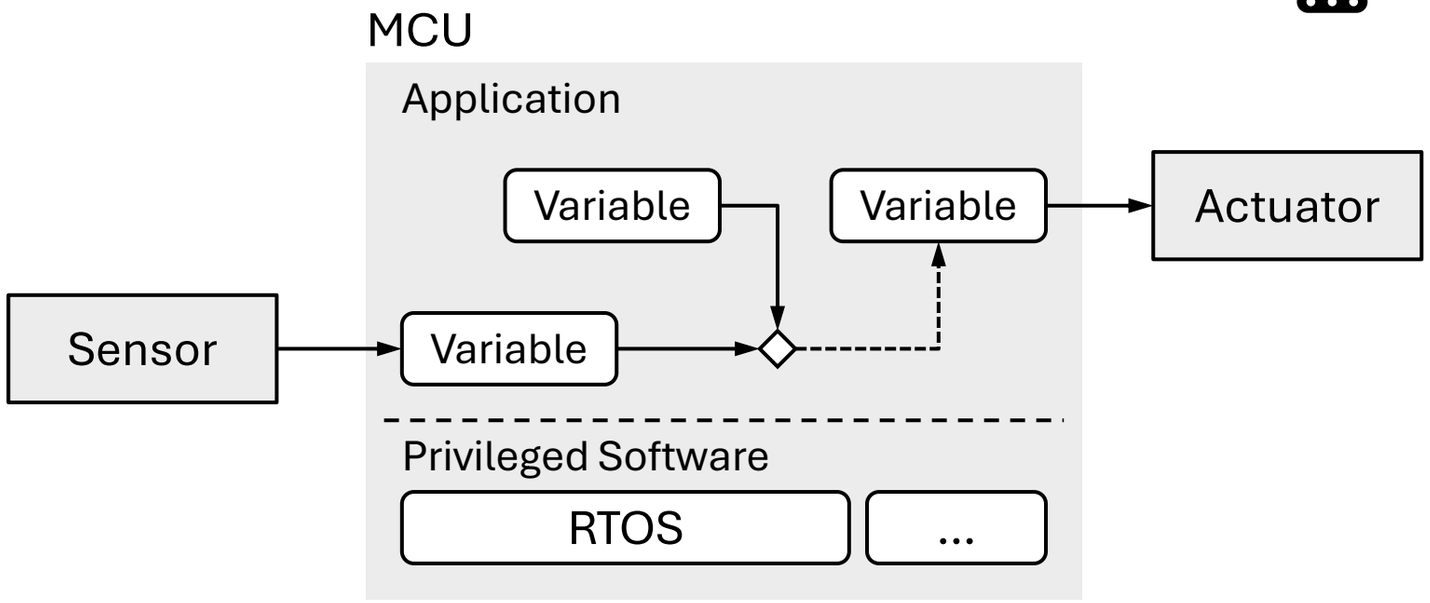


Data Security of MCU

Microcontroller units are used in critical fields

- Healthcare
- Industrial automation
- Autonomous driving vehicles

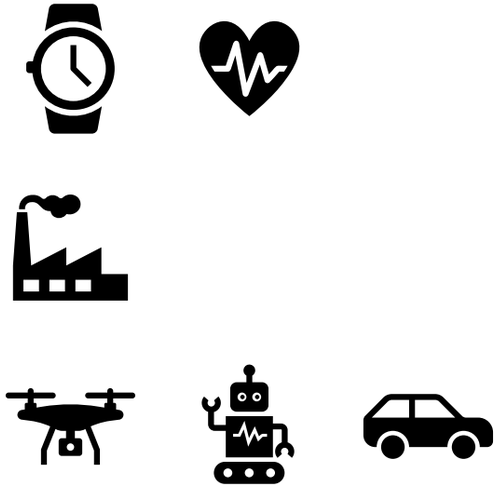


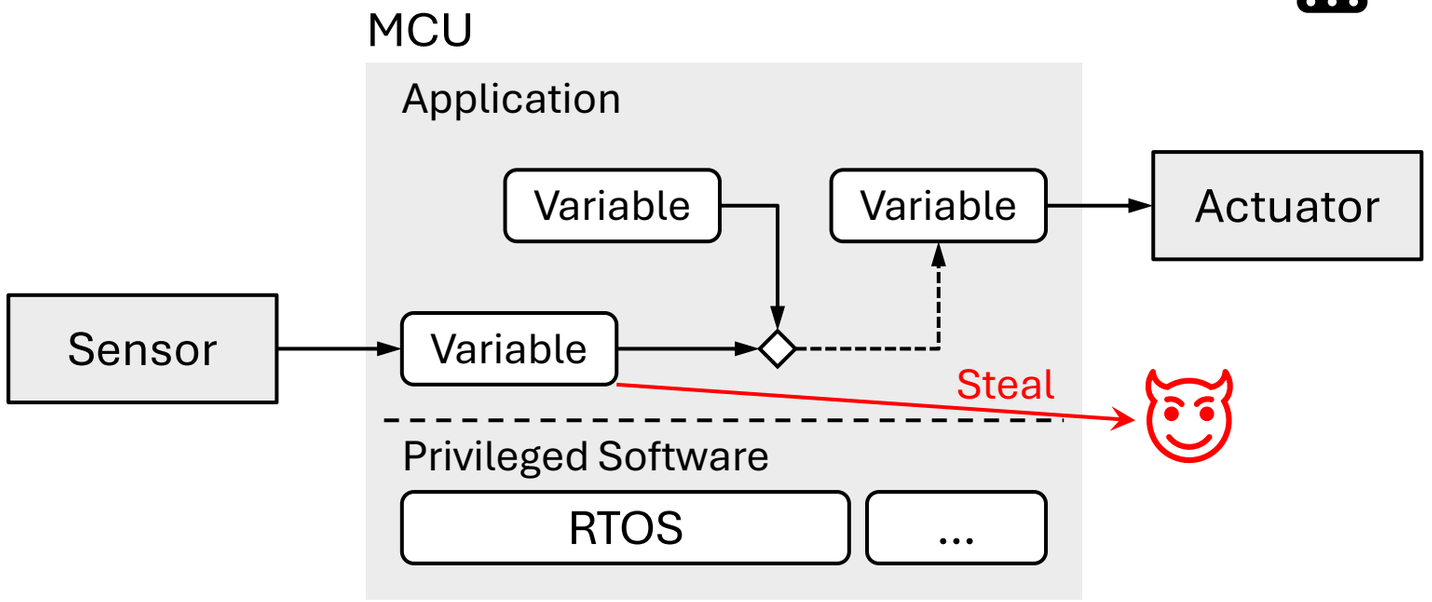


Data Security of MCU

Microcontroller units are used in critical fields

- Healthcare
- Industrial automation
- Autonomous driving vehicles

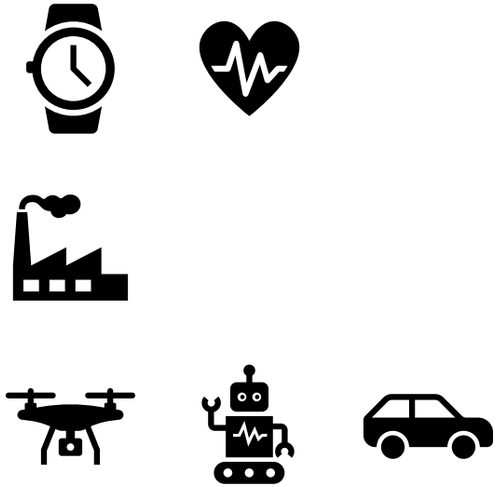


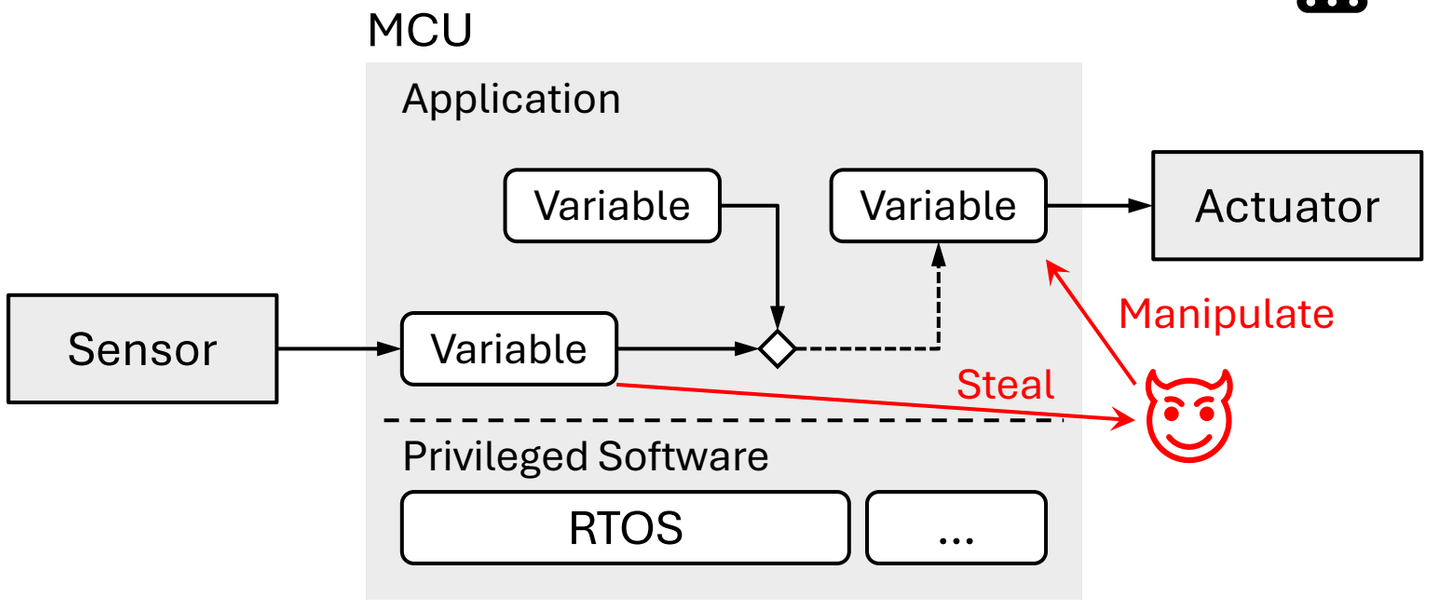


Data Security of MCU

Microcontroller units are used in critical fields

- Healthcare
- Industrial automation
- Autonomous driving vehicles

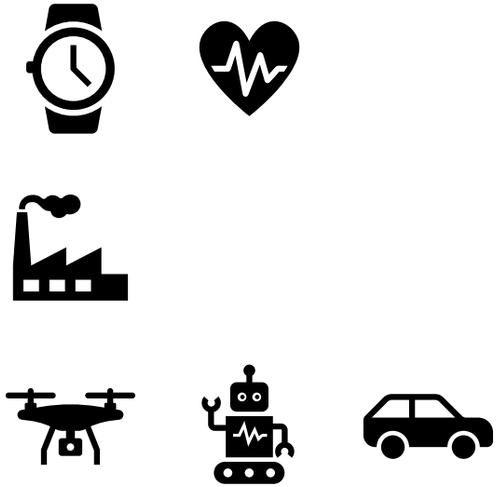


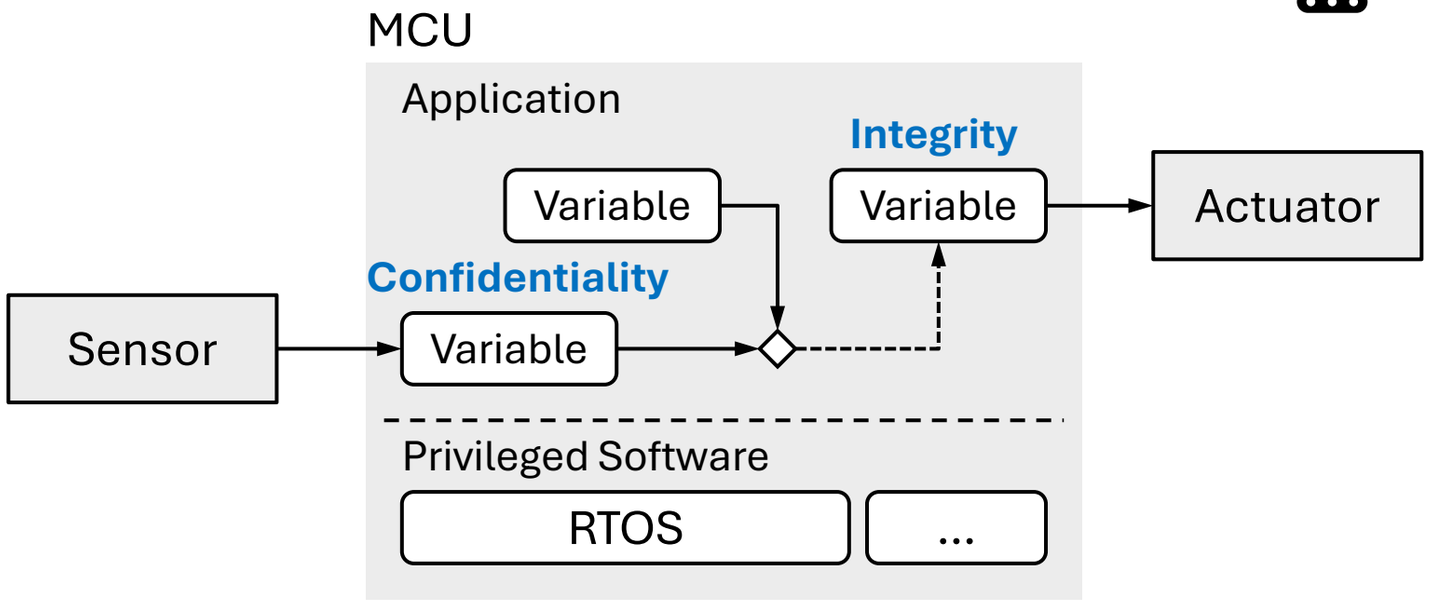


Data Security of MCU

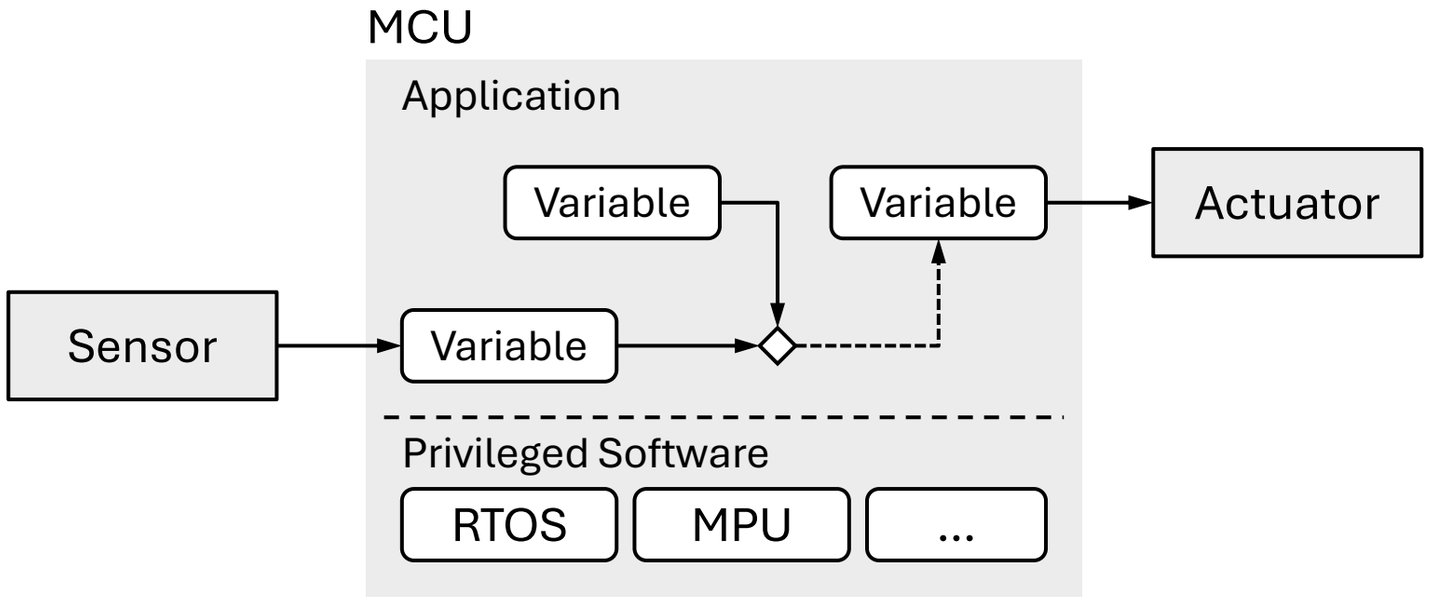
Microcontroller units are used in critical fields

- Healthcare
- Industrial automation
- Autonomous driving vehicles



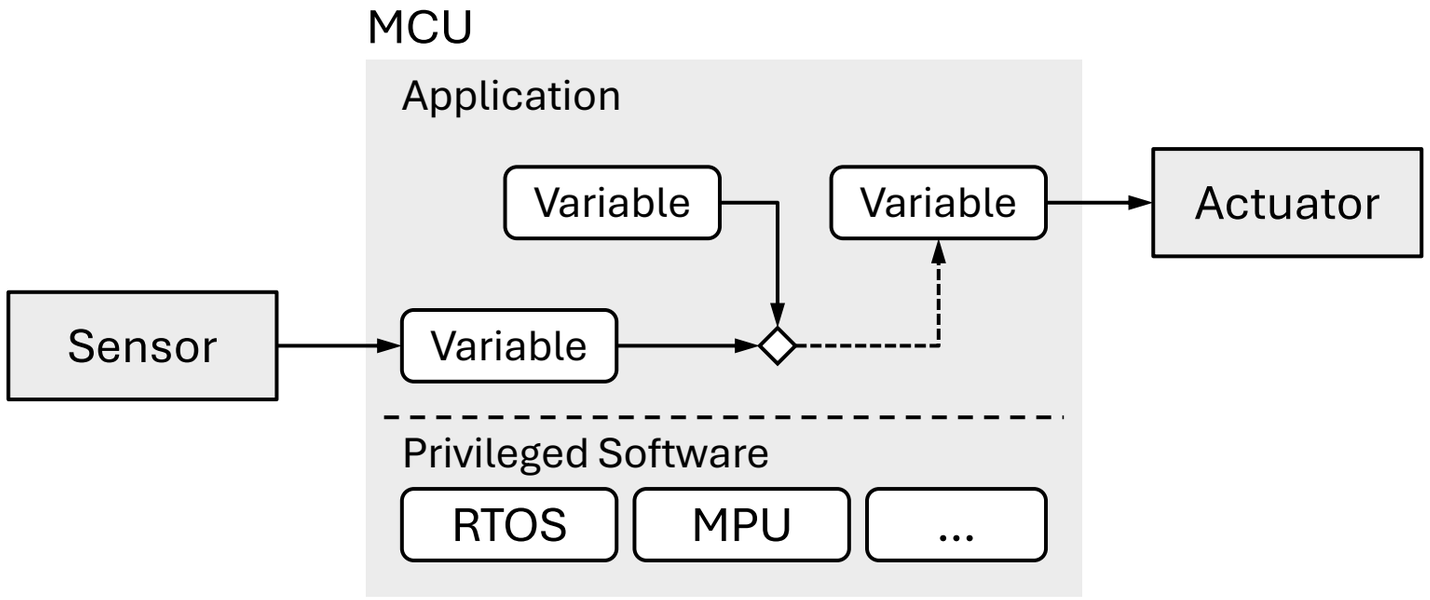


Protection against Strong Adversaries



Protection against Strong Adversaries

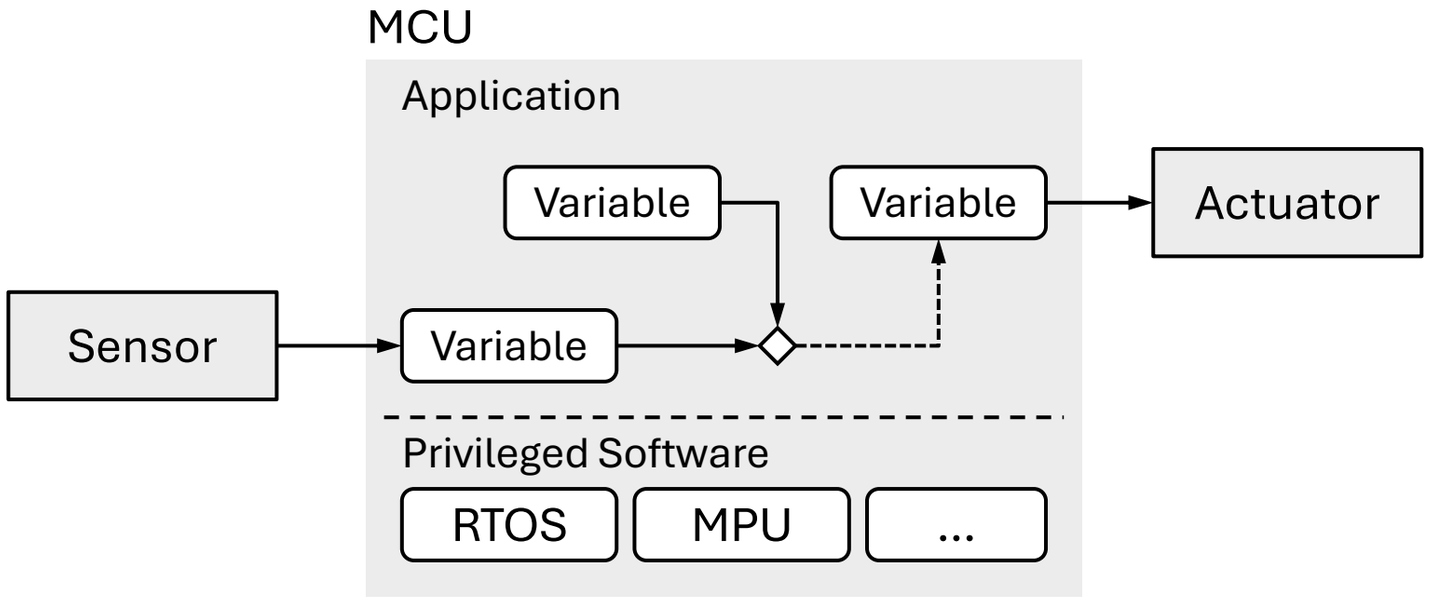
Memory Protection Unit (MPU):



Protection against Strong Adversaries

Memory Protection Unit (MPU):

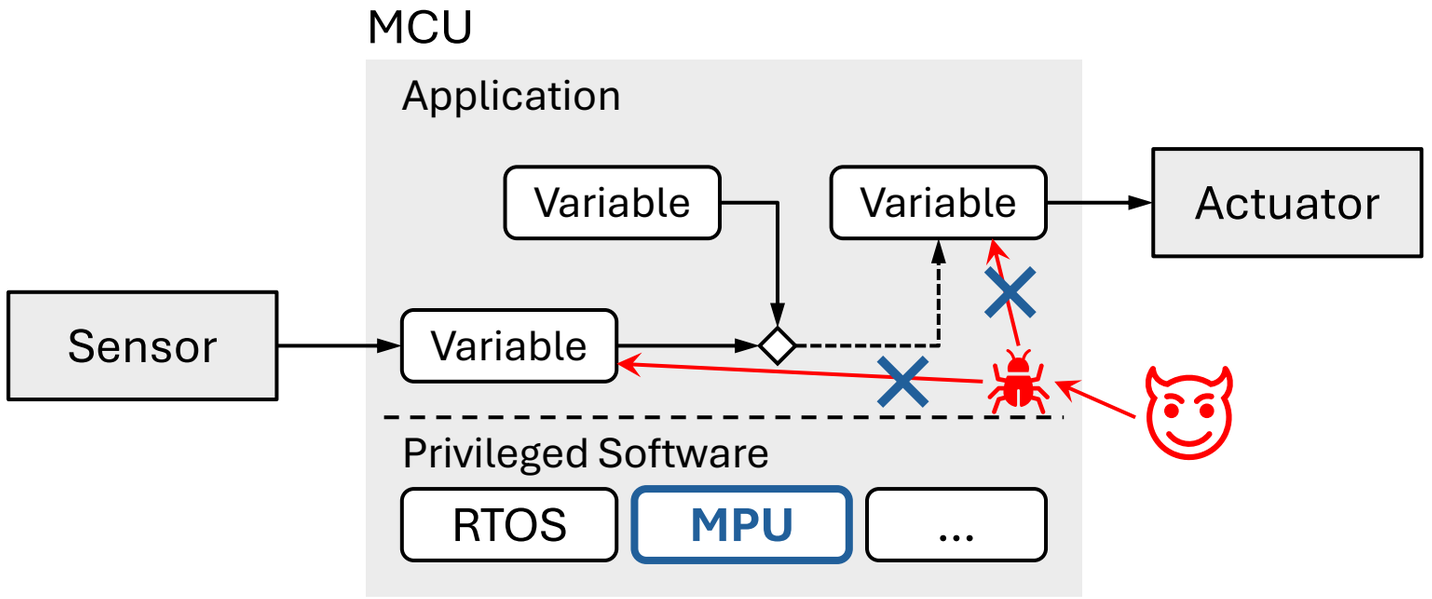
- Hardware extension: protect memory regions by defining access permissions



Protection against Strong Adversaries

Memory Protection Unit (MPU):

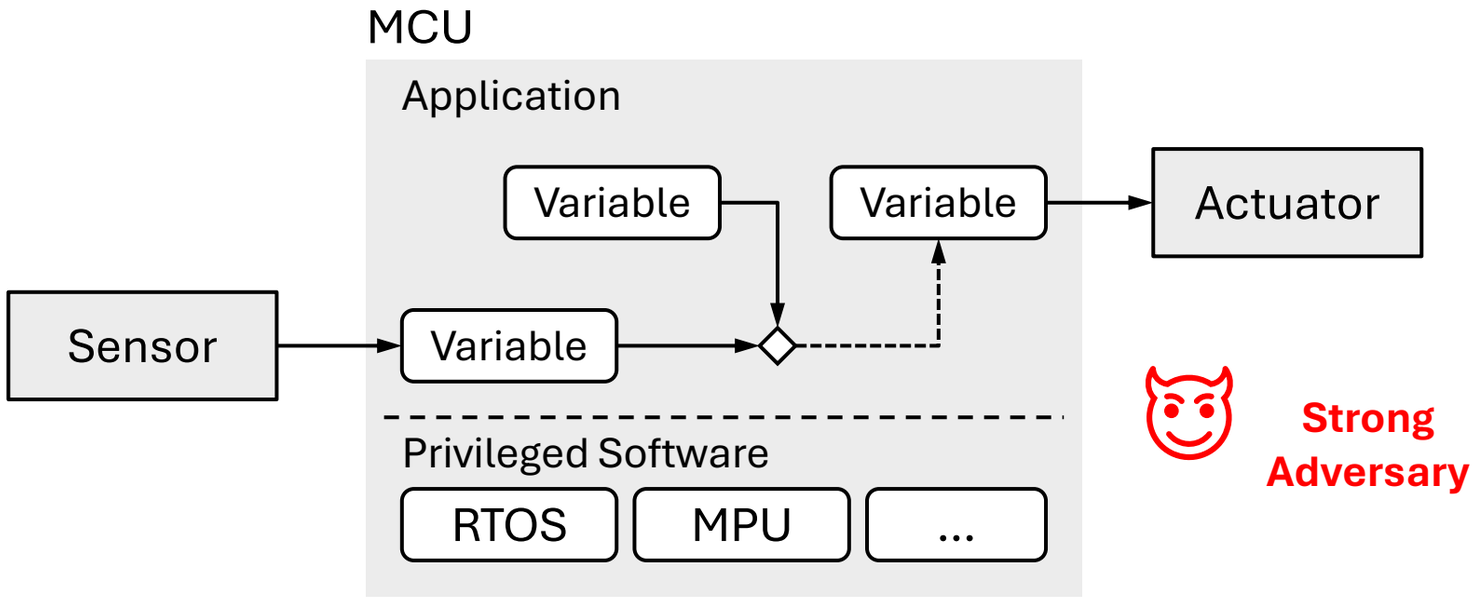
- Hardware extension: protect memory regions by defining access permissions



Protection against Strong Adversaries

Memory Protection Unit (MPU):

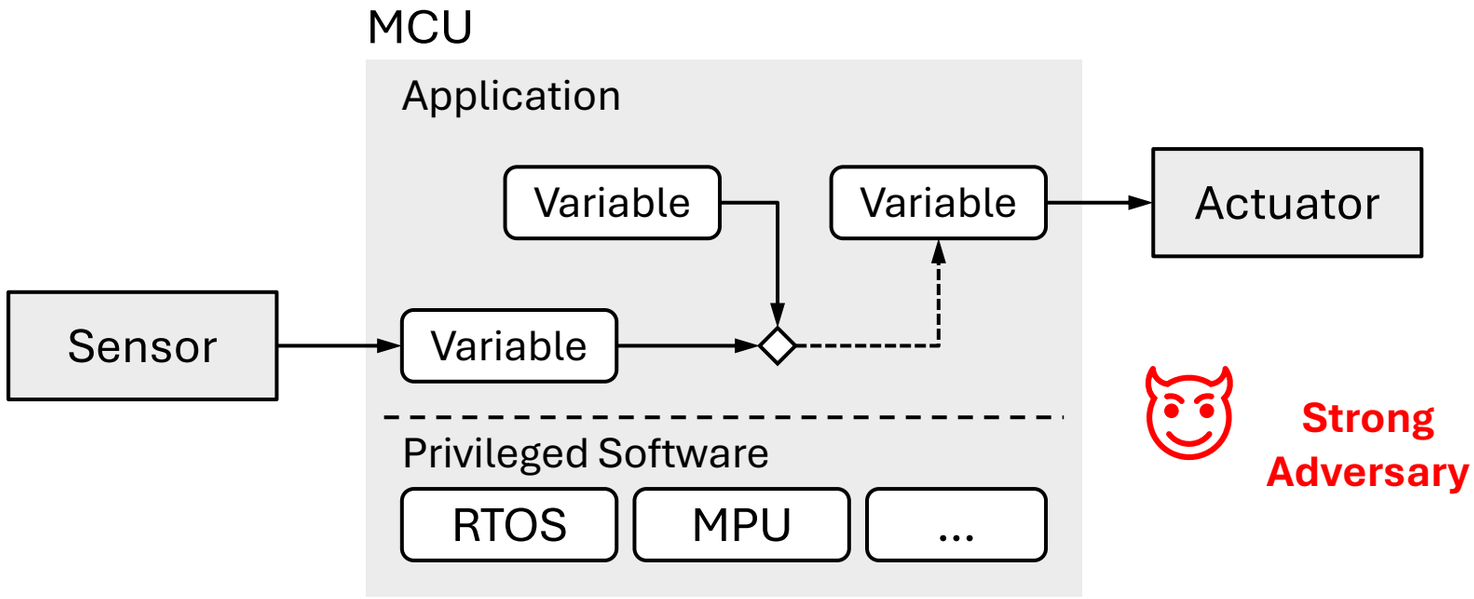
- Hardware extension: protect memory regions by defining access permissions
- Existing MPU-based protection is ineffective against strong adversaries



Protection against Strong Adversaries

Memory Protection Unit (MPU):

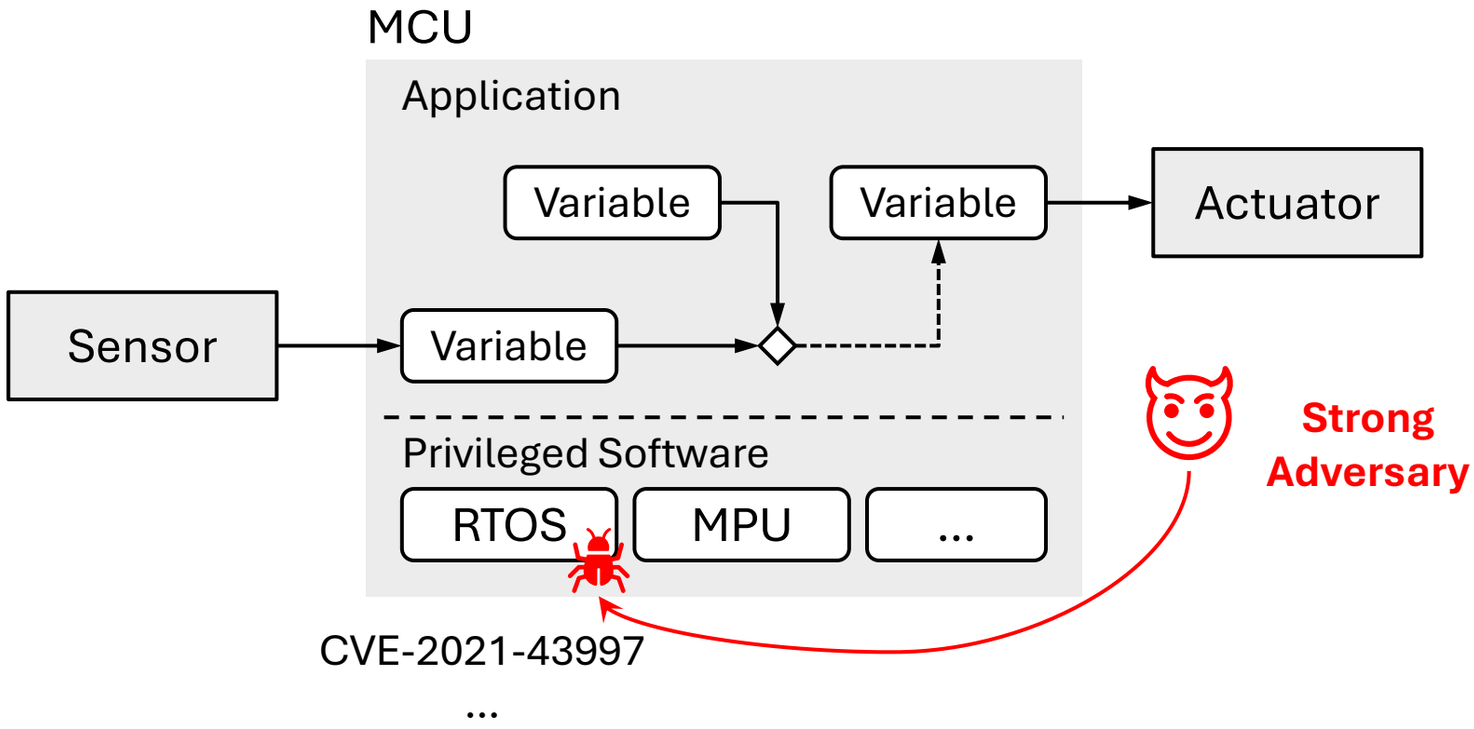
- Hardware extension: protect memory regions by defining access permissions
- Existing MPU-based protection is ineffective against strong adversaries
- MPU itself needs to be configured in privileged mode



Protection against Strong Adversaries

Memory Protection Unit (MPU):

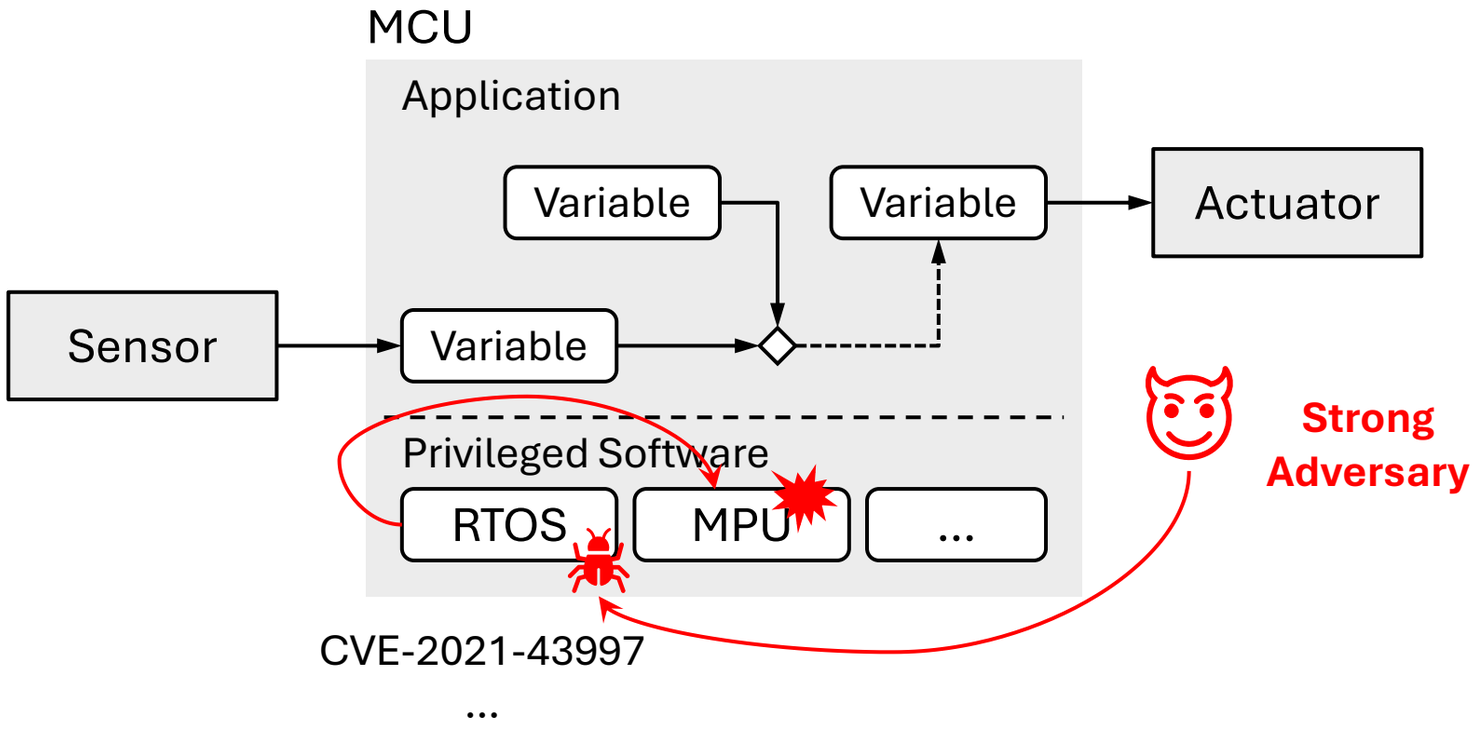
- Hardware extension: protect memory regions by defining access permissions
- Existing MPU-based protection is ineffective against strong adversaries
- MPU itself needs to be configured in privileged mode



Protection against Strong Adversaries

Memory Protection Unit (MPU):

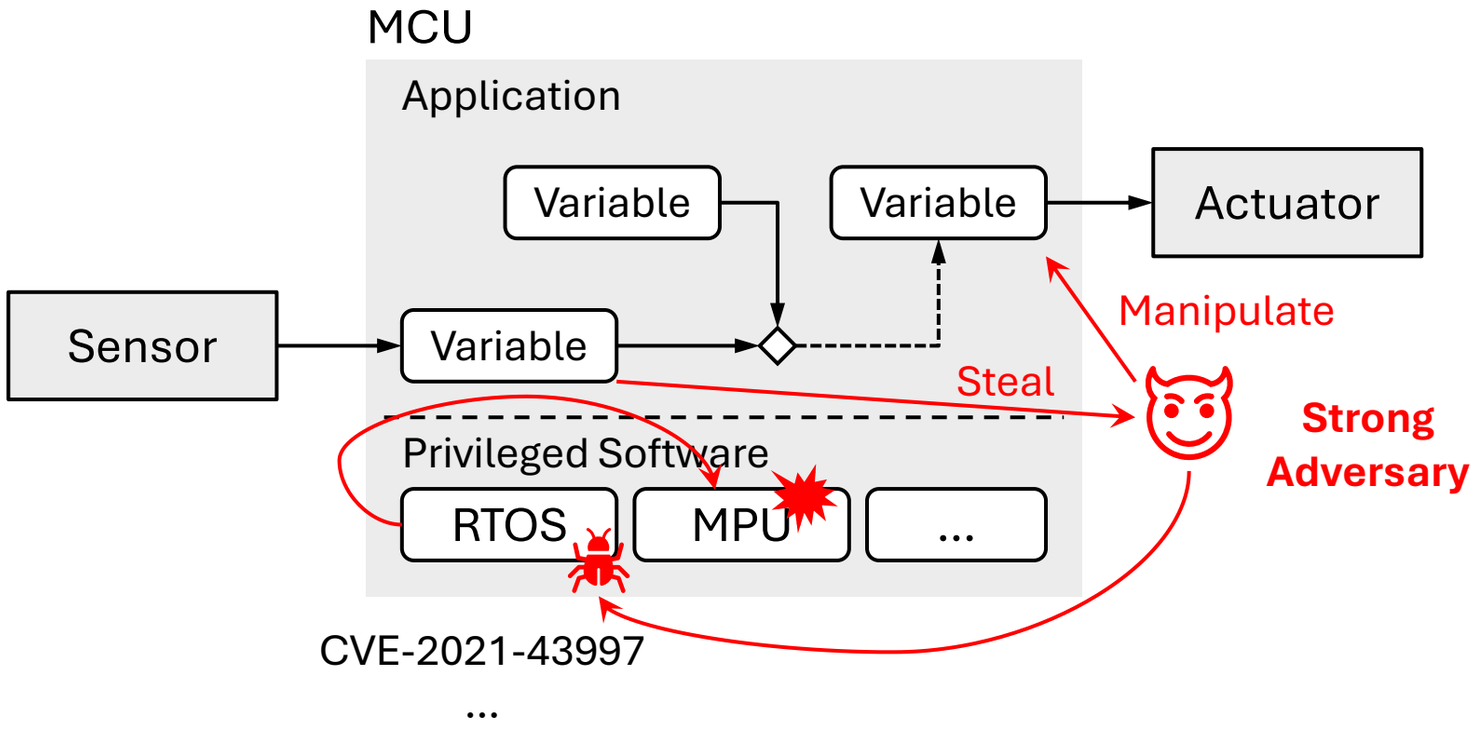
- Hardware extension: protect memory regions by defining access permissions
- Existing MPU-based protection is ineffective against strong adversaries
- MPU itself needs to be configured in privileged mode



Protection against Strong Adversaries

Memory Protection Unit (MPU):

- Hardware extension: protect memory regions by defining access permissions
- Existing MPU-based protection is ineffective against strong adversaries
- MPU itself needs to be configured in privileged mode



ARM TrustZone for MCU Data Protection

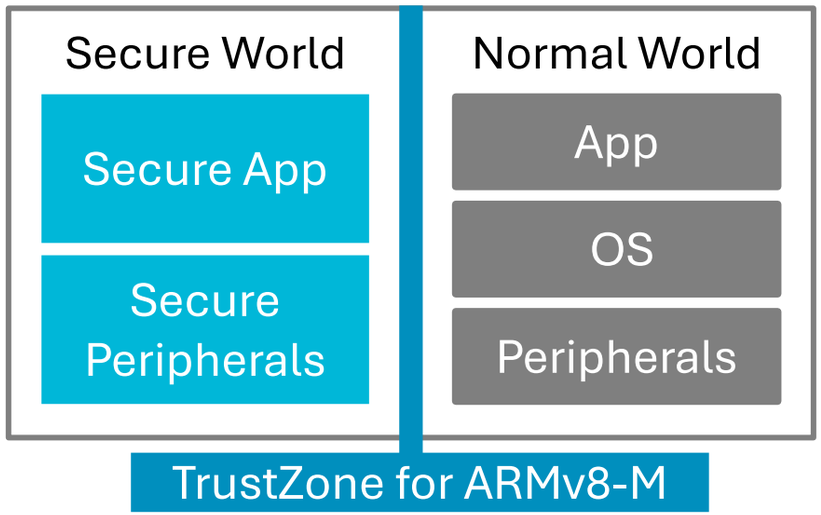
ARM TrustZone for MCU Data Protection

Goal: protect integrity and confidentiality of data in MCU against strong adversaries using ARM TrustZone

ARM TrustZone for MCU Data Protection

Goal: protect integrity and confidentiality of data in MCU against strong adversaries using ARM TrustZone

ARM TrustZone

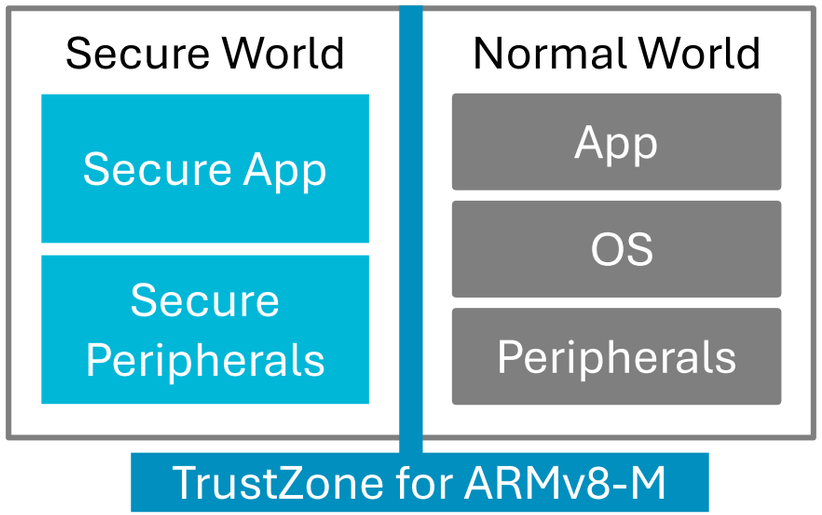


ARM TrustZone for MCU Data Protection

Goal: protect integrity and confidentiality of data in MCU against strong adversaries using ARM TrustZone

ARM TrustZone

- The secure world of ARM TrustZone for MCU provides a Trusted Execution Environment

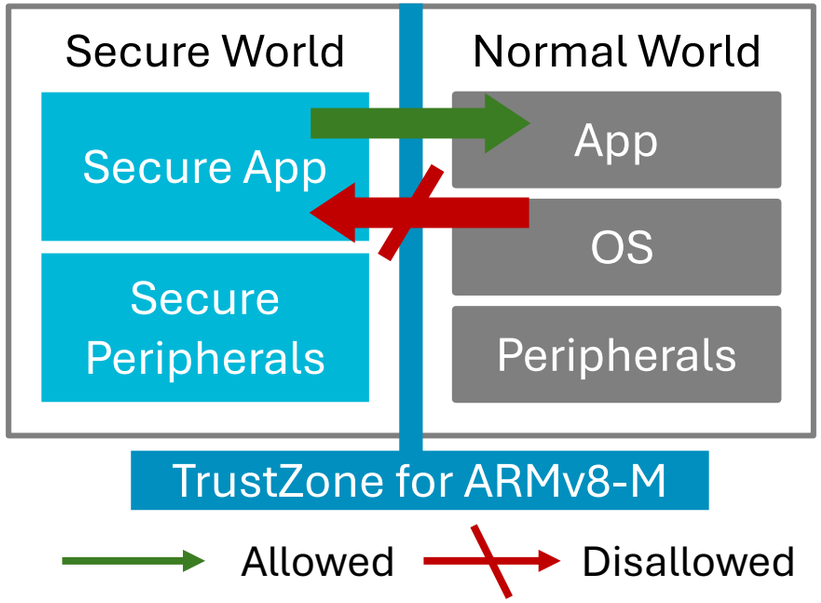


ARM TrustZone for MCU Data Protection

Goal: protect integrity and confidentiality of data in MCU against strong adversaries using ARM TrustZone

ARM TrustZone

- The secure world of ARM TrustZone for MCU provides a Trusted Execution Environment
- Protect against strong adversaries in normal world

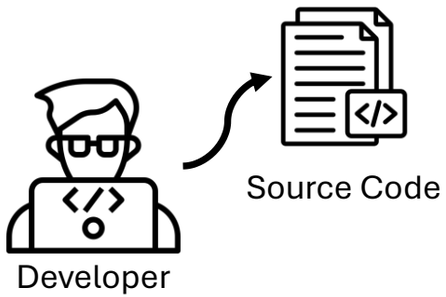


Overview of TZ-DATASHIELD

Overview of TZ-DATASHIELD

Overview of TZ-DATASHIELD

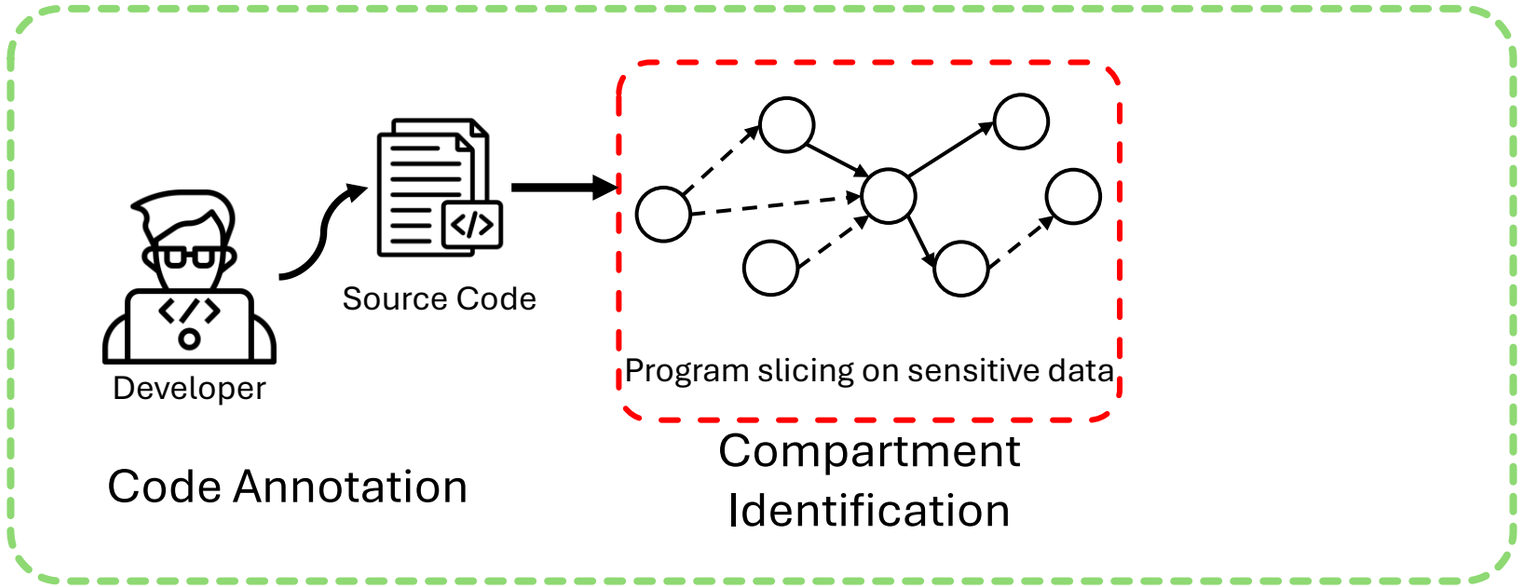
- Annotate sensitive data (variables and peripherals)



Code Annotation

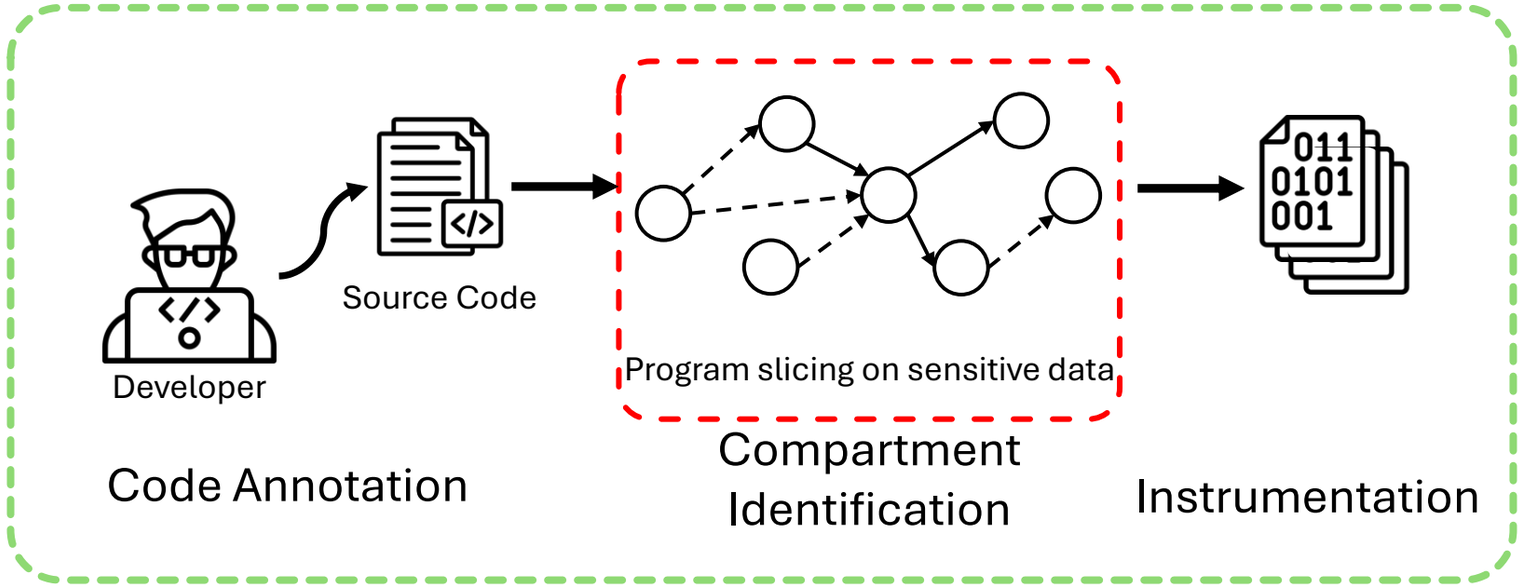
Overview of TZ-DATASHIELD

- Annotate sensitive data (variables and peripherals)
- Identify compartments based on data flow of sensitive data



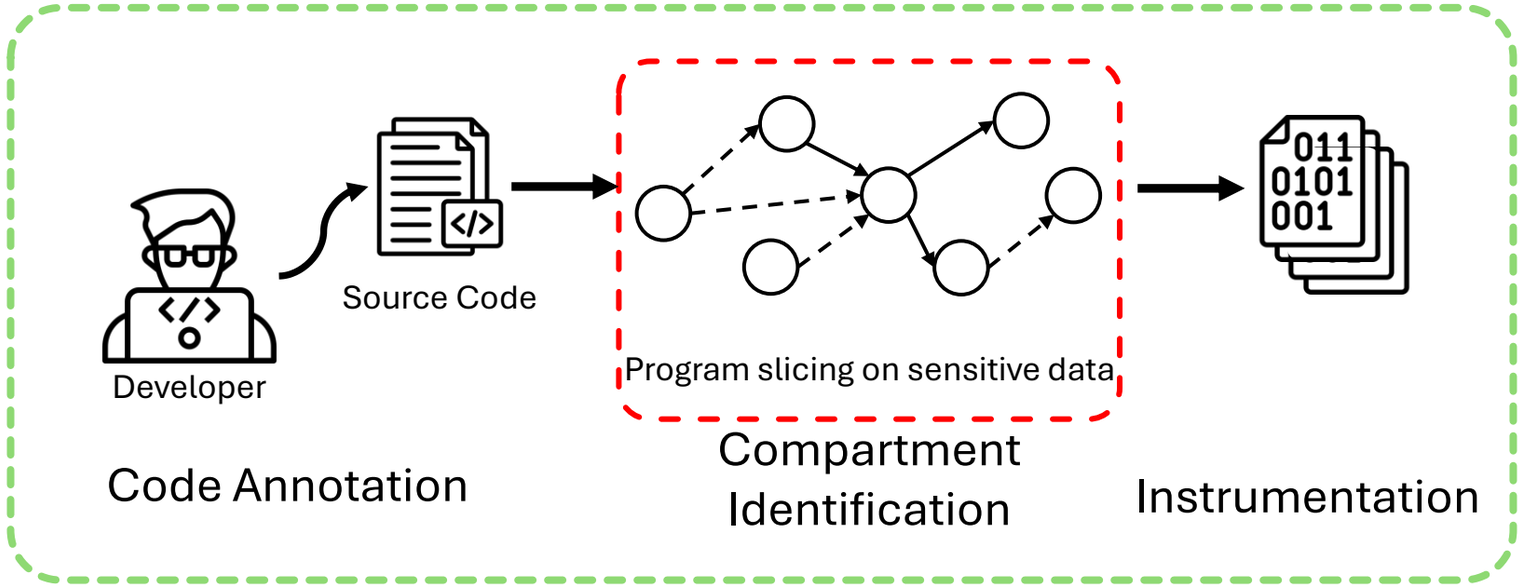
Overview of TZ-DATASHIELD

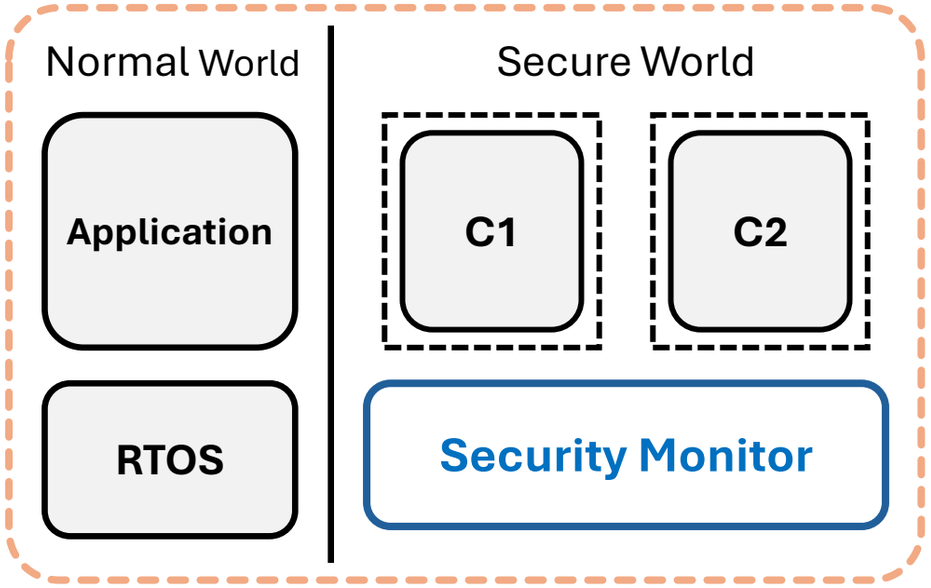
- Annotate sensitive data (variables and peripherals)
- Identify compartments based on data flow of sensitive data
- Instrument LLVM IR code to isolate compartments



Overview of TZ-DATASHIELD

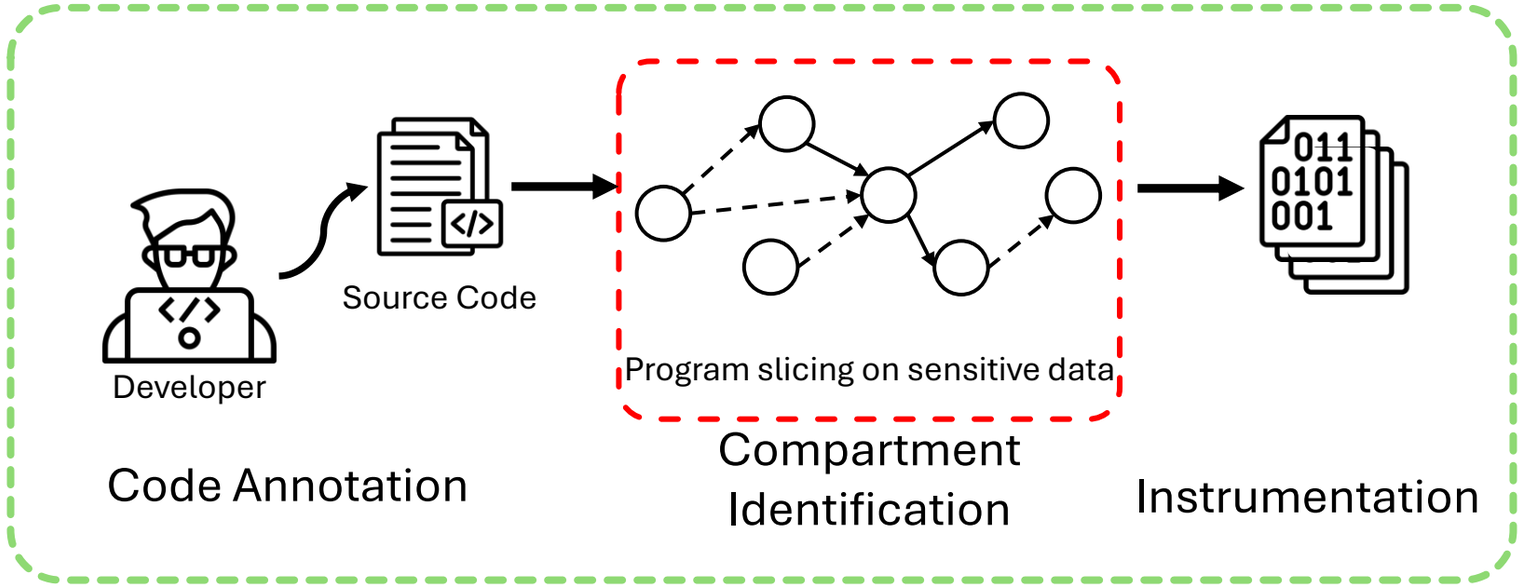
- Annotate sensitive data (variables and peripherals)
- Identify compartments based on data flow of sensitive data
- Instrument LLVM IR code to isolate compartments
- Enforce isolation by security monitor during runtime

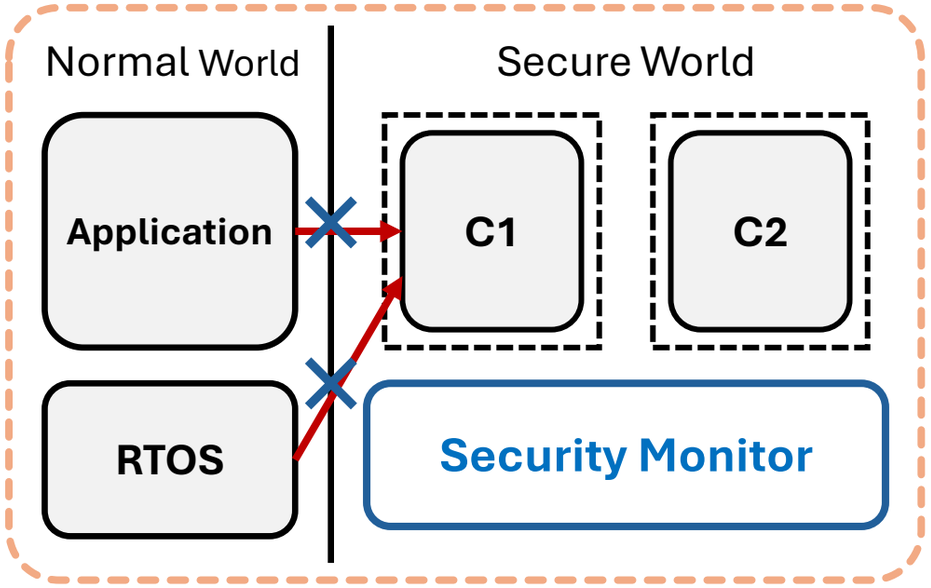




Overview of TZ-DATASHIELD

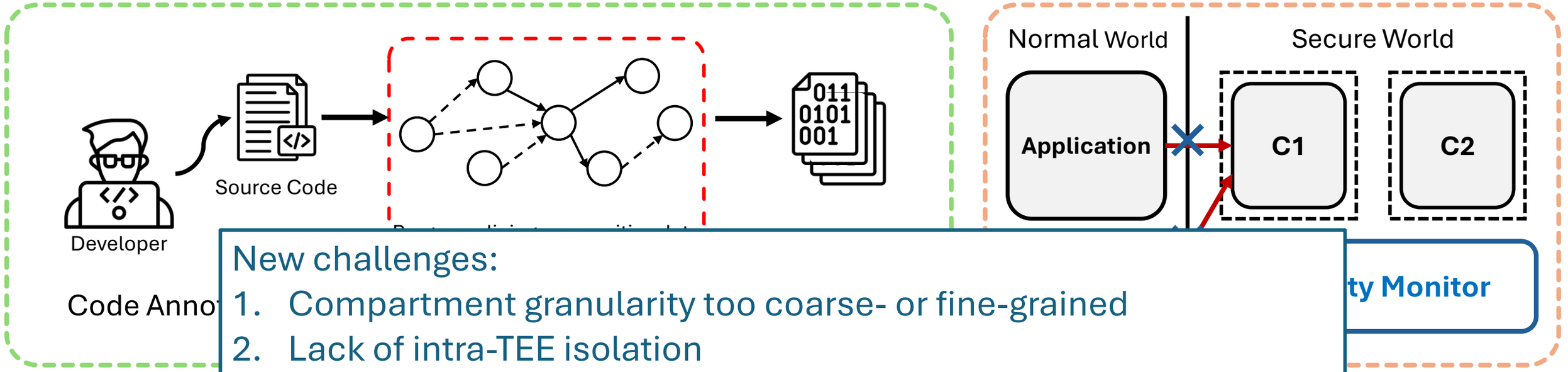
- Annotate sensitive data (variables and peripherals)
- Identify compartments based on data flow of sensitive data
- Instrument LLVM IR code to isolate compartments
- Enforce isolation by security monitor during runtime





Overview of TZ-DATASHIELD

- Annotate sensitive data (variables and peripherals)
- Identify compartments based on data flow of sensitive data
- Instrument LLVM IR code to isolate compartments
- Enforce isolation by security monitor during runtime



New challenges:

1. Compartment granularity too coarse- or fine-grained
2. Lack of intra-TEE isolation
3. Protection on shared data/peripheral between compartments

Challenge 1: Compartment Granularity

Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

Challenge 1: Compartment Granularity

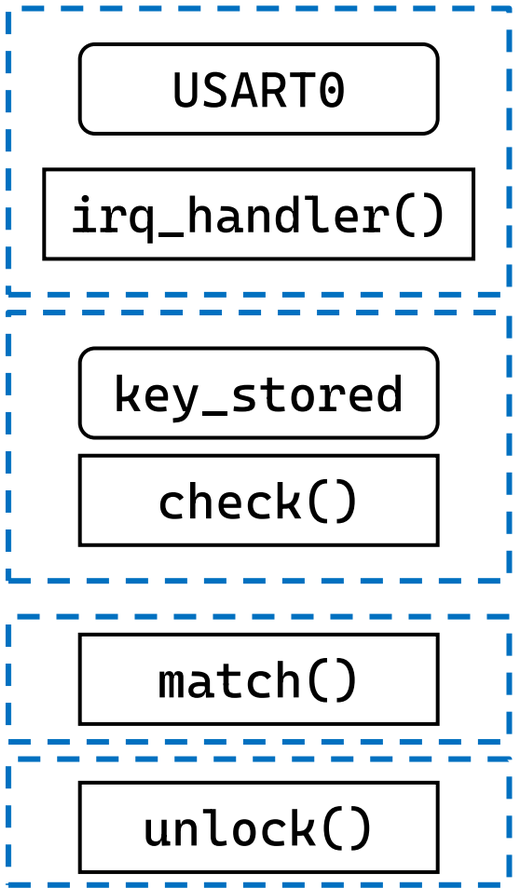
Existing MPU-based compartmentalization approaches

- **Function-level**: Fine-grained isolation per function [SEC'18]

Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

- **Function-level**: Fine-grained isolation per function [SEC'18]

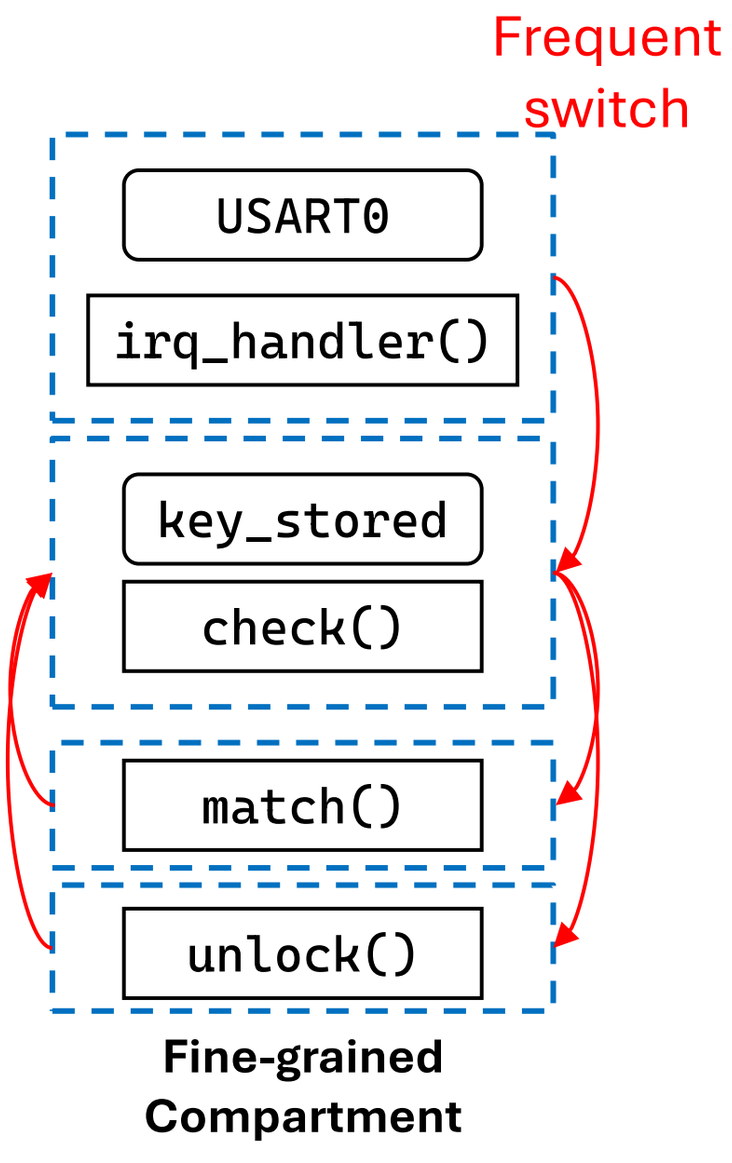


Fine-grained Compartment

Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

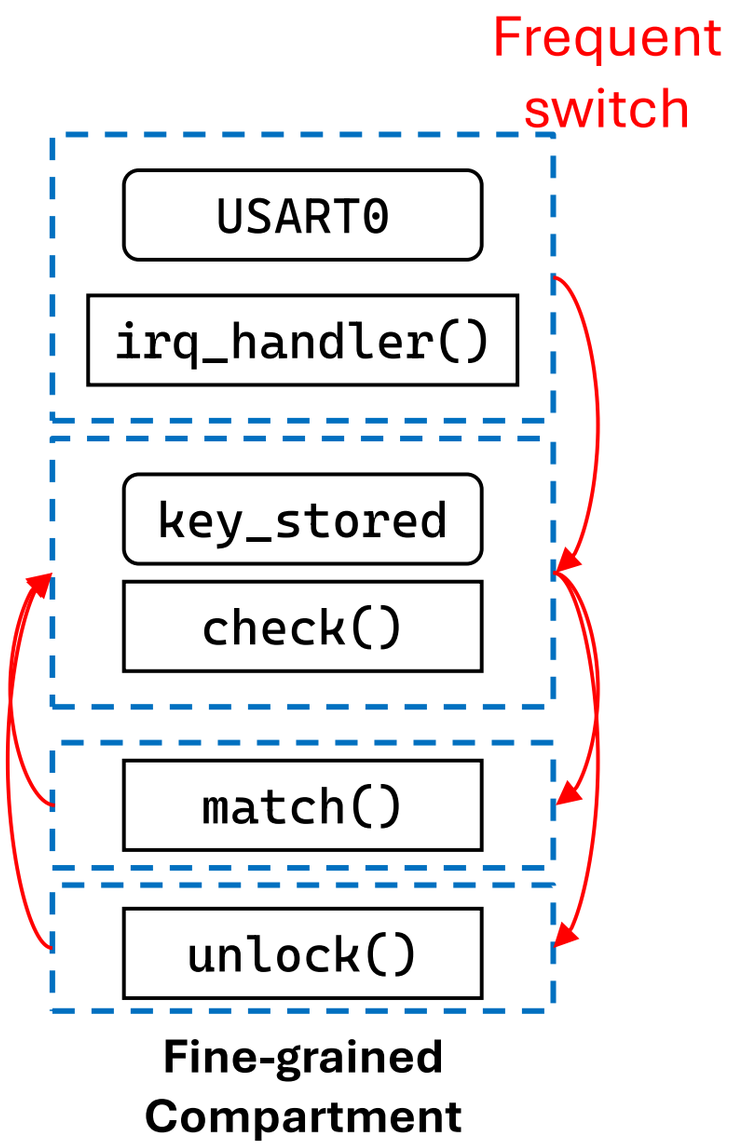
- **Function-level**: Fine-grained isolation per function [SEC'18]



Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

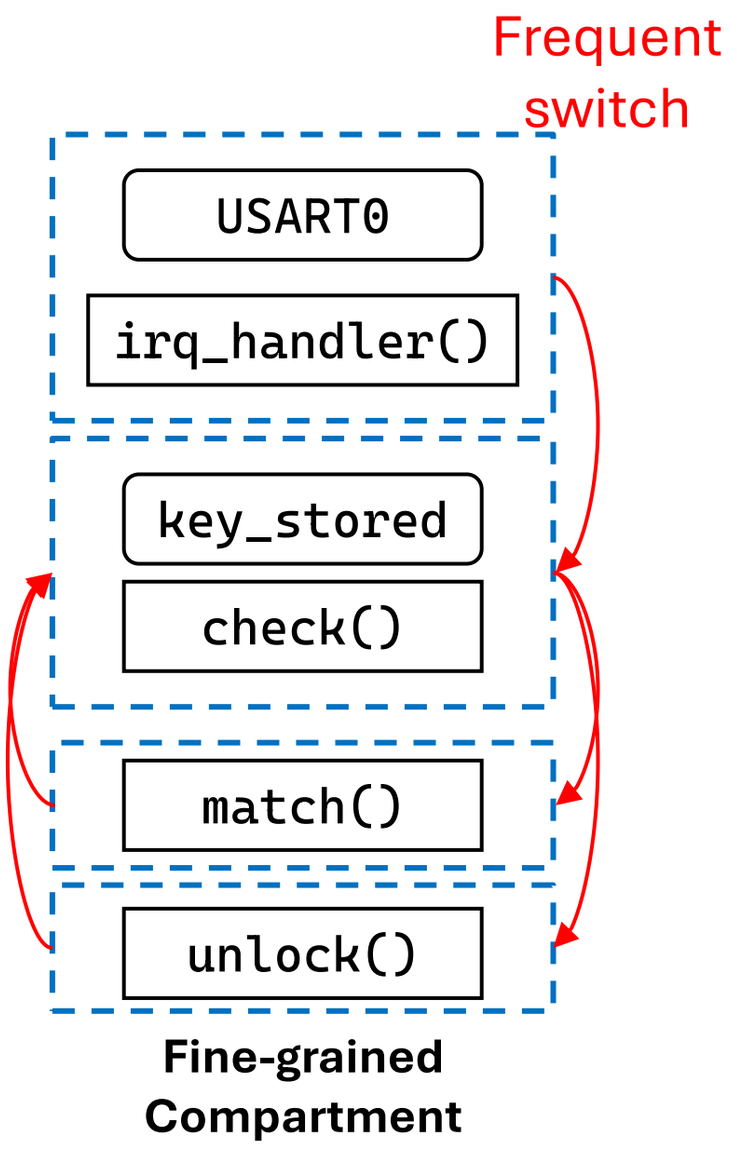
- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]



Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

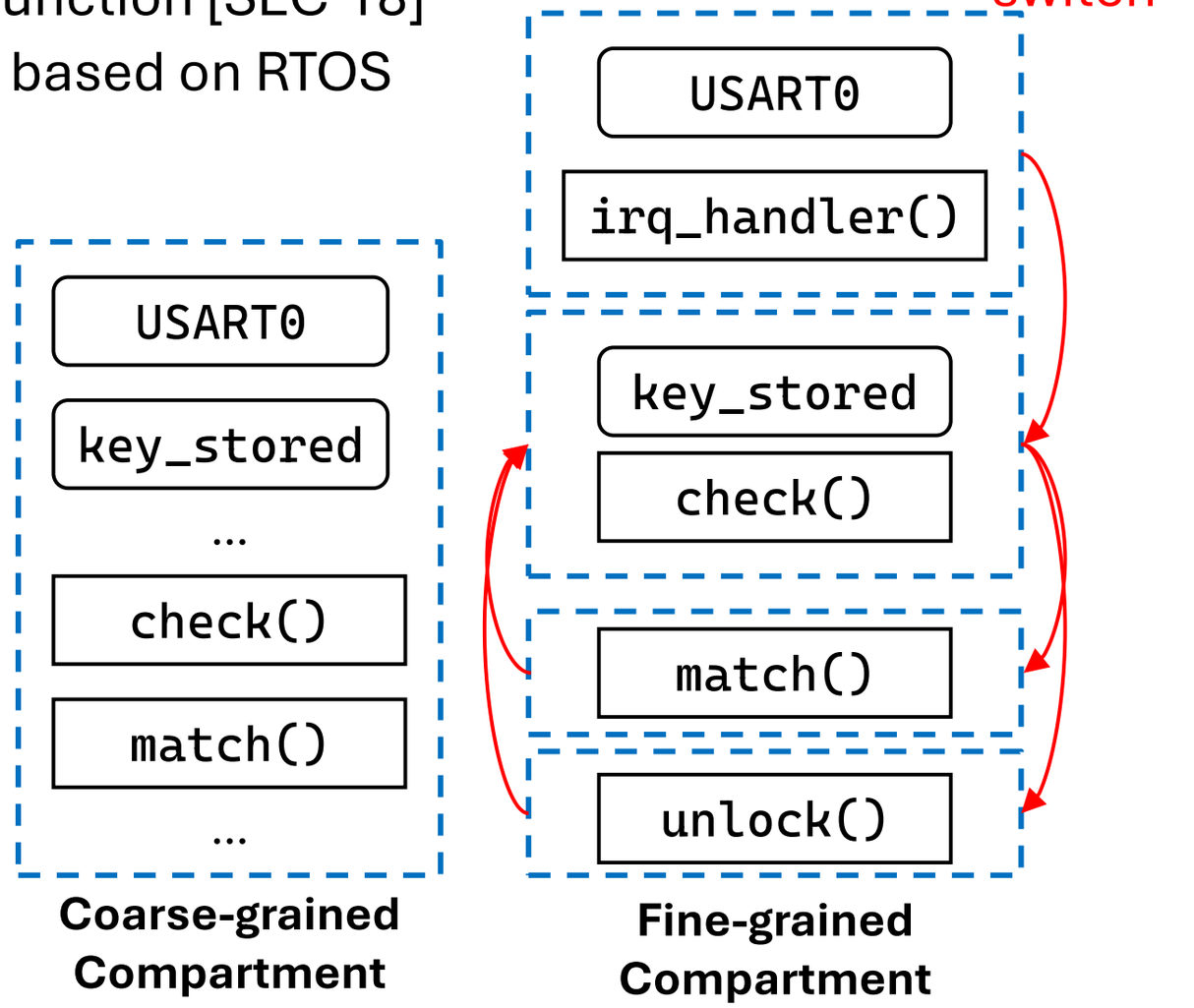
- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]
- **File/component-level**: Isolation by software components like libraries and peripheral drivers [SP'23]



Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

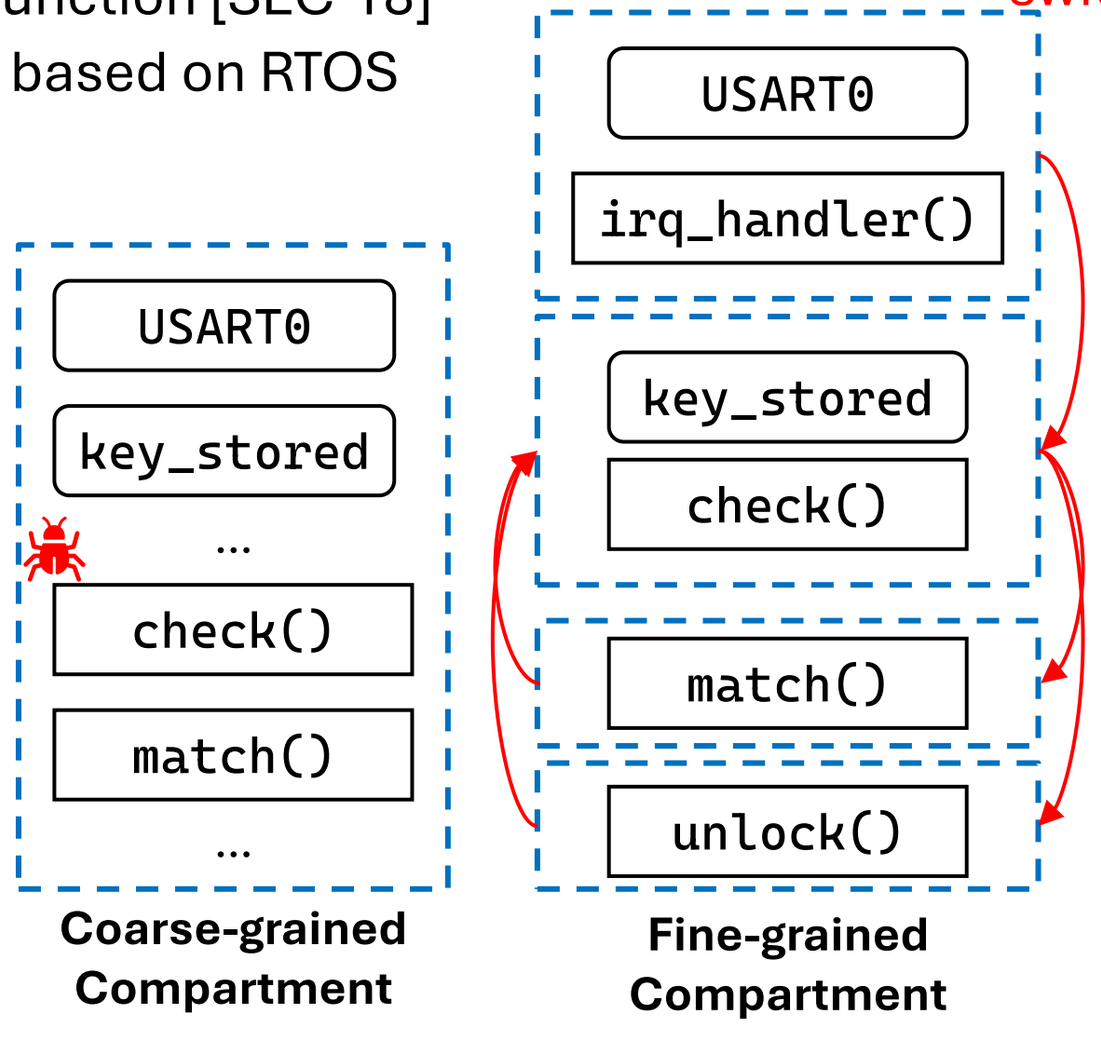
- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]
- **File/component-level**: Isolation by software components like libraries and peripheral drivers [SP'23]



Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

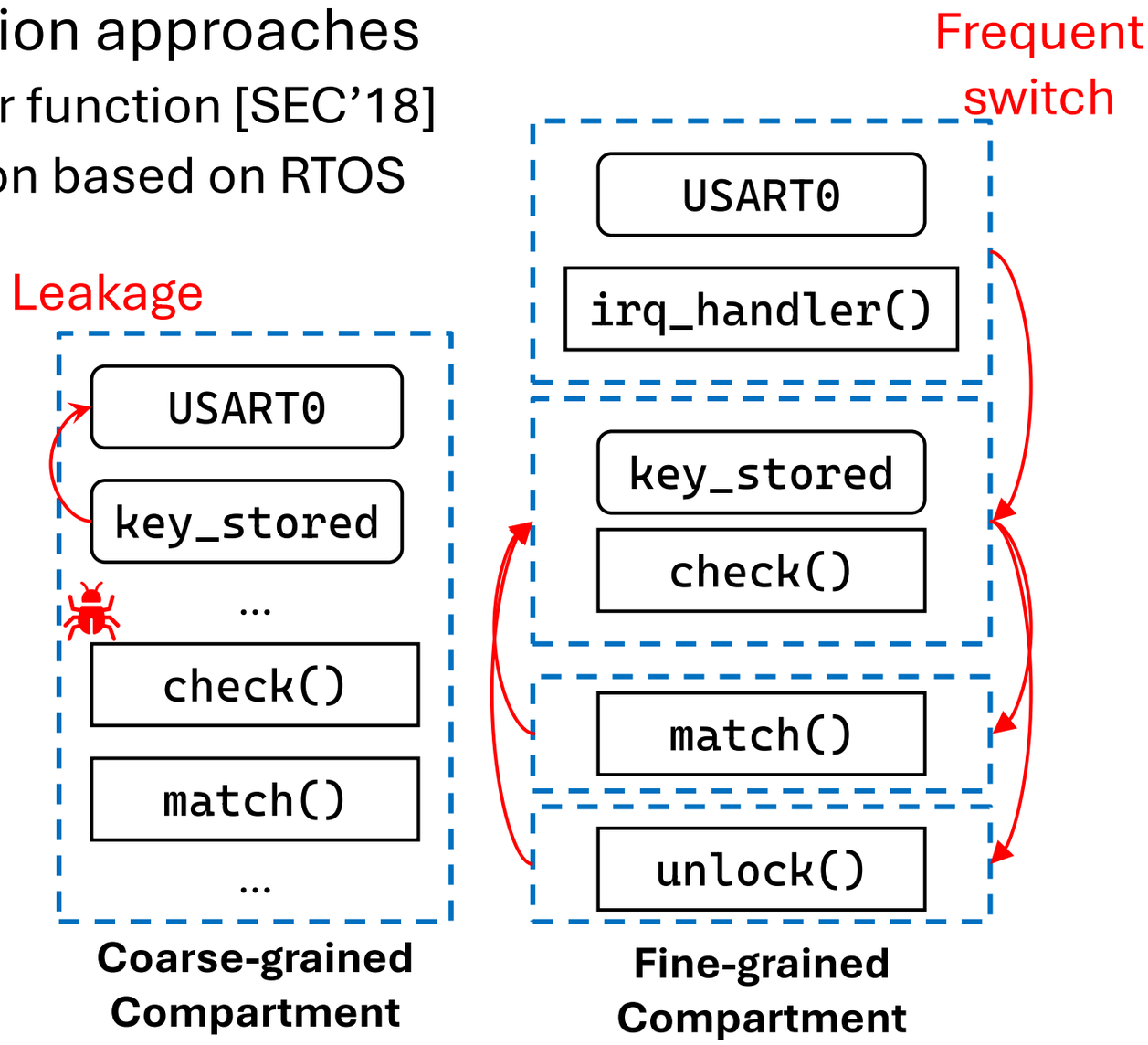
- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]
- **File/component-level**: Isolation by software components like libraries and peripheral drivers [SP'23]



Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]
- **File/component-level**: Isolation by software components like libraries and peripheral drivers [SP'23]

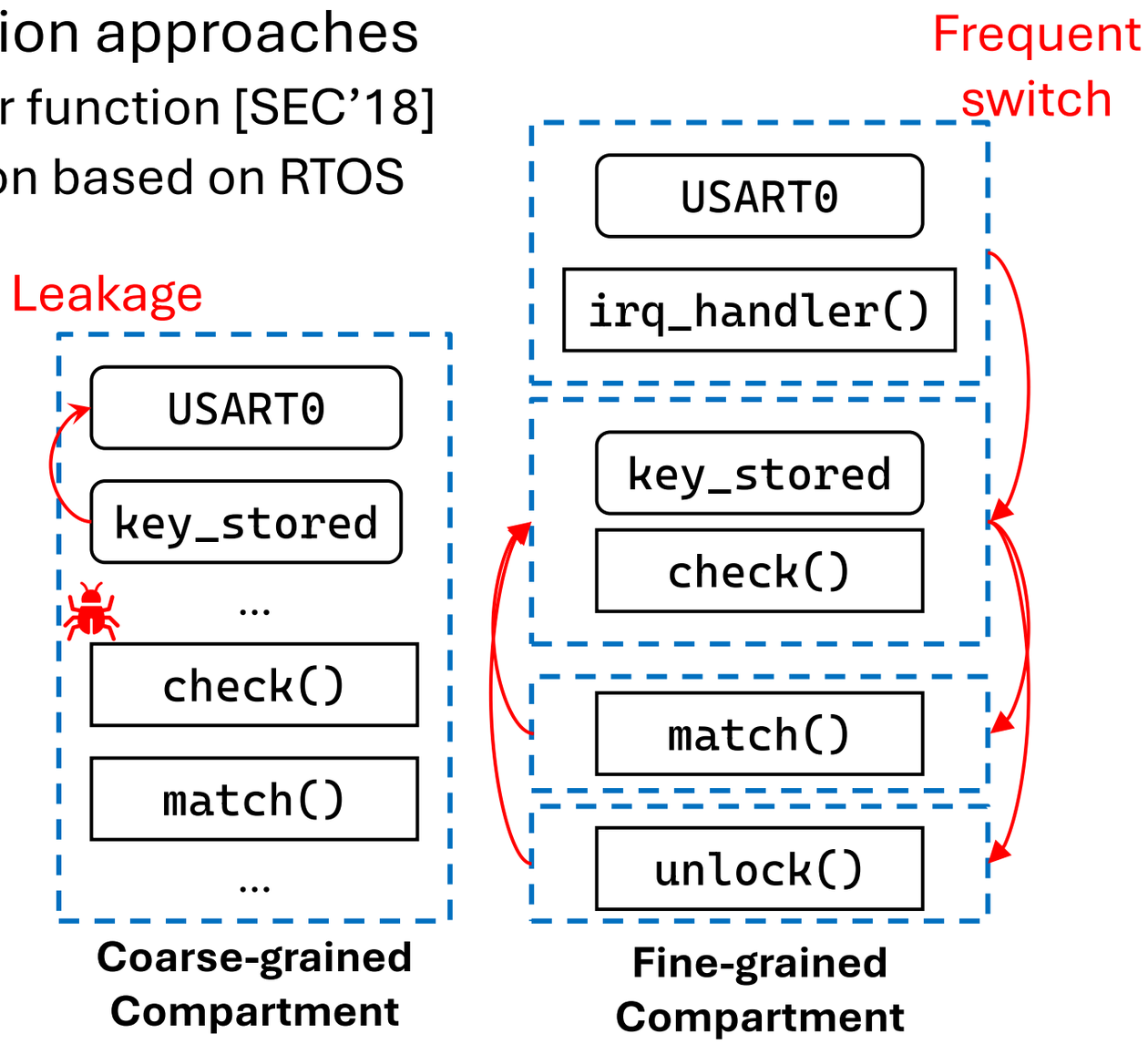


Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]
- **File/component-level**: Isolation by software components like libraries and peripheral drivers [SP'23]

Not designed for data protection:



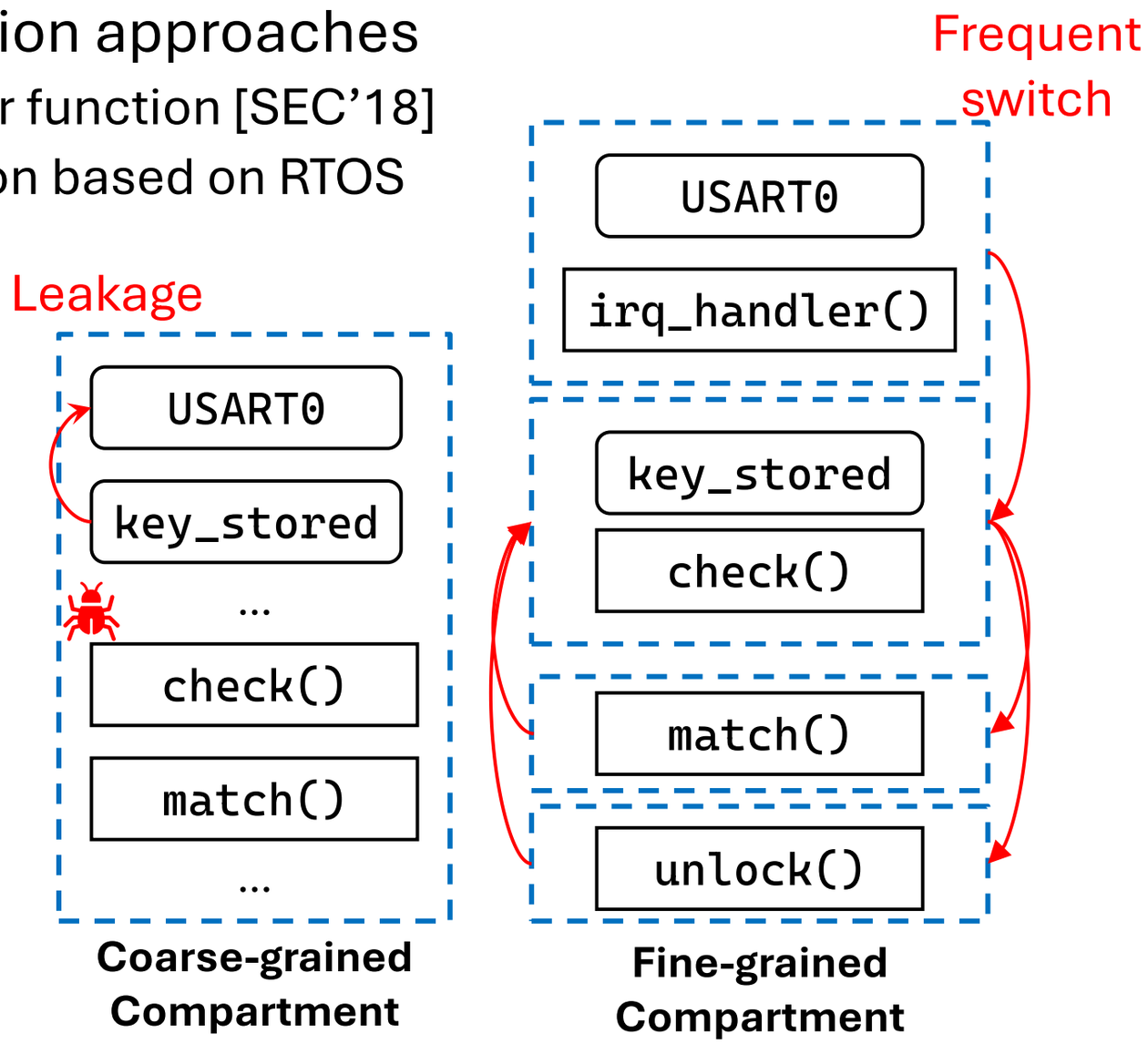
Challenge 1: Compartment Granularity

Existing MPU-based compartmentalization approaches

- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]
- **File/component-level**: Isolation by software components like libraries and peripheral drivers [SP'23]

Not designed for data protection:

- Either too coarse- or fine-grained



Challenge 1: Compartment Granularity

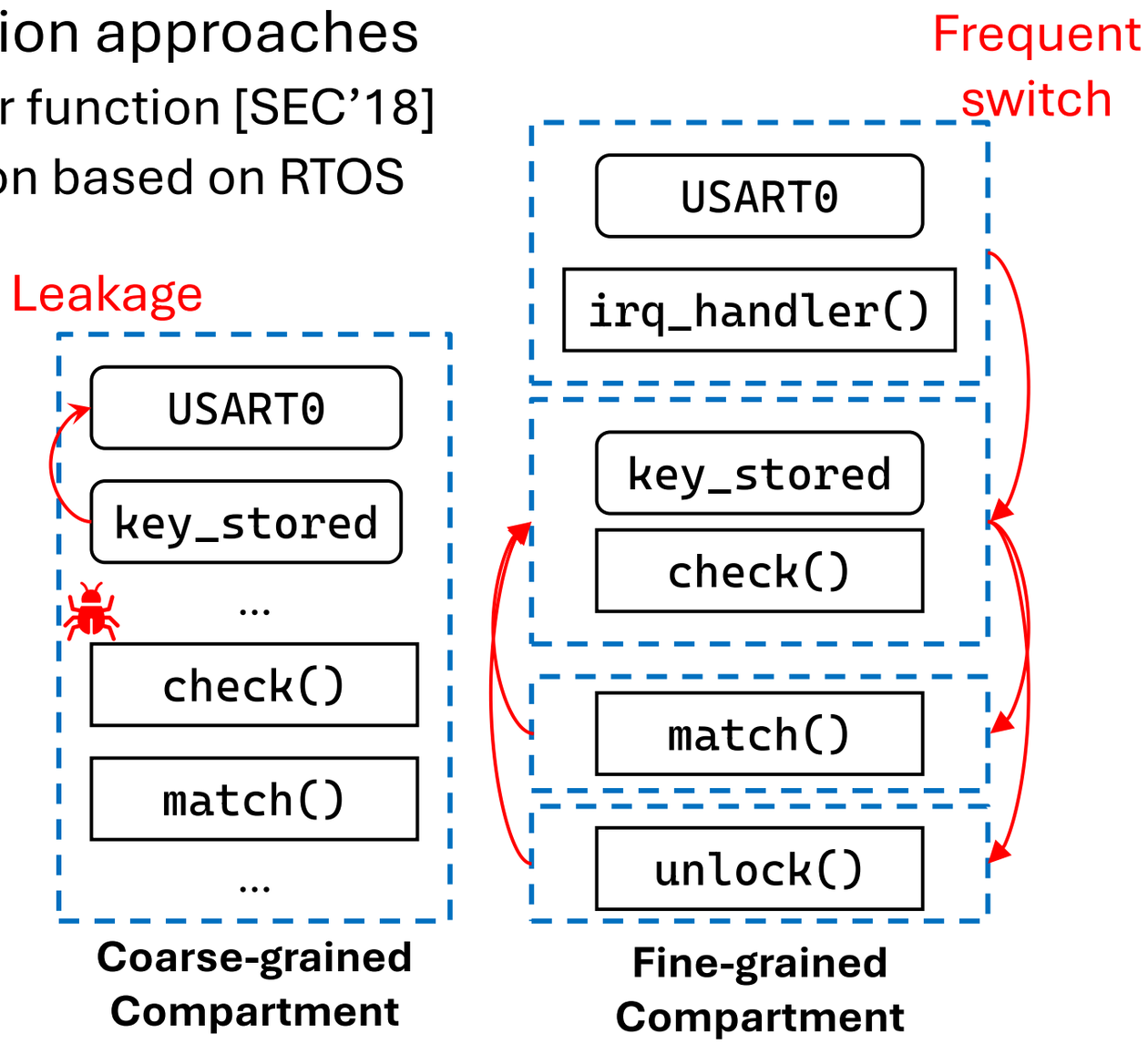
Existing MPU-based compartmentalization approaches

- **Function-level**: Fine-grained isolation per function [SEC'18]
- **RTOS thread-level**: Compartmentalization based on RTOS threads [SP'23, NDSS'18]
- **File/component-level**: Isolation by software components like libraries and peripheral drivers [SP'23]

Not designed for data protection:

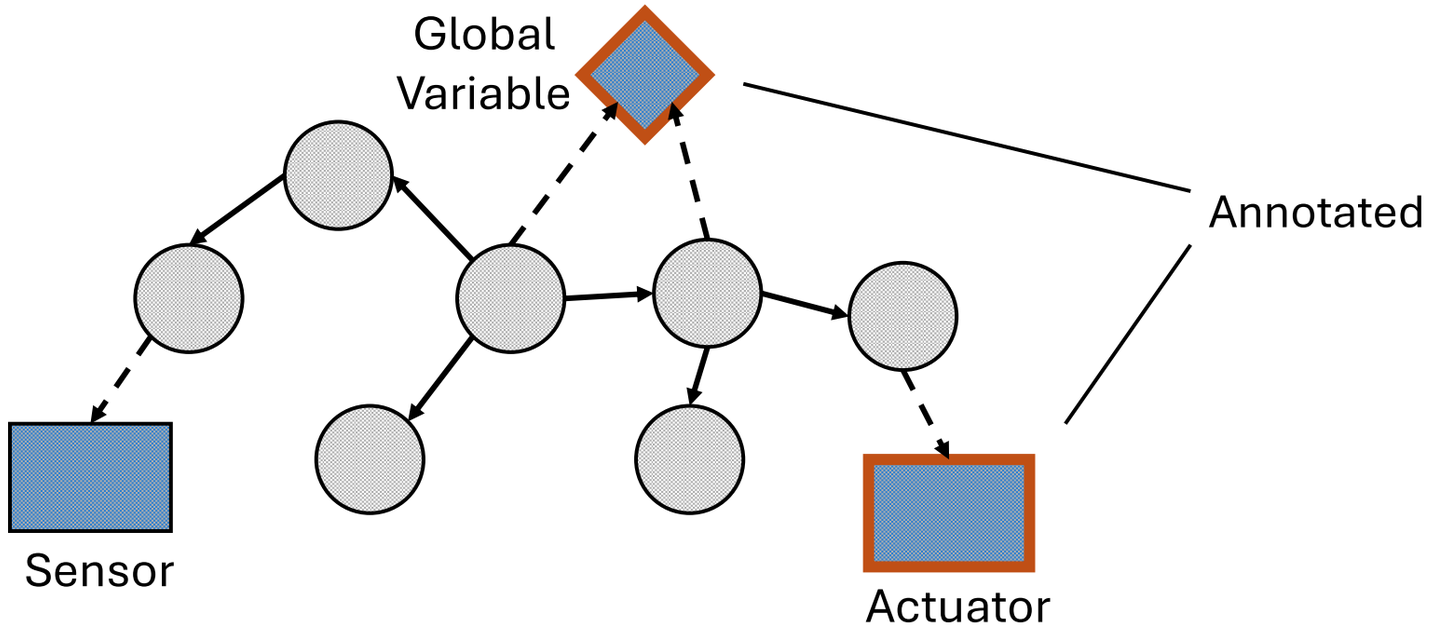
- Either too coarse- or fine-grained

Our solution: **Sensitive Data Flow** (SDF) Compartmentalization



Solution 1: Sensitive Data Flow-based Compartment

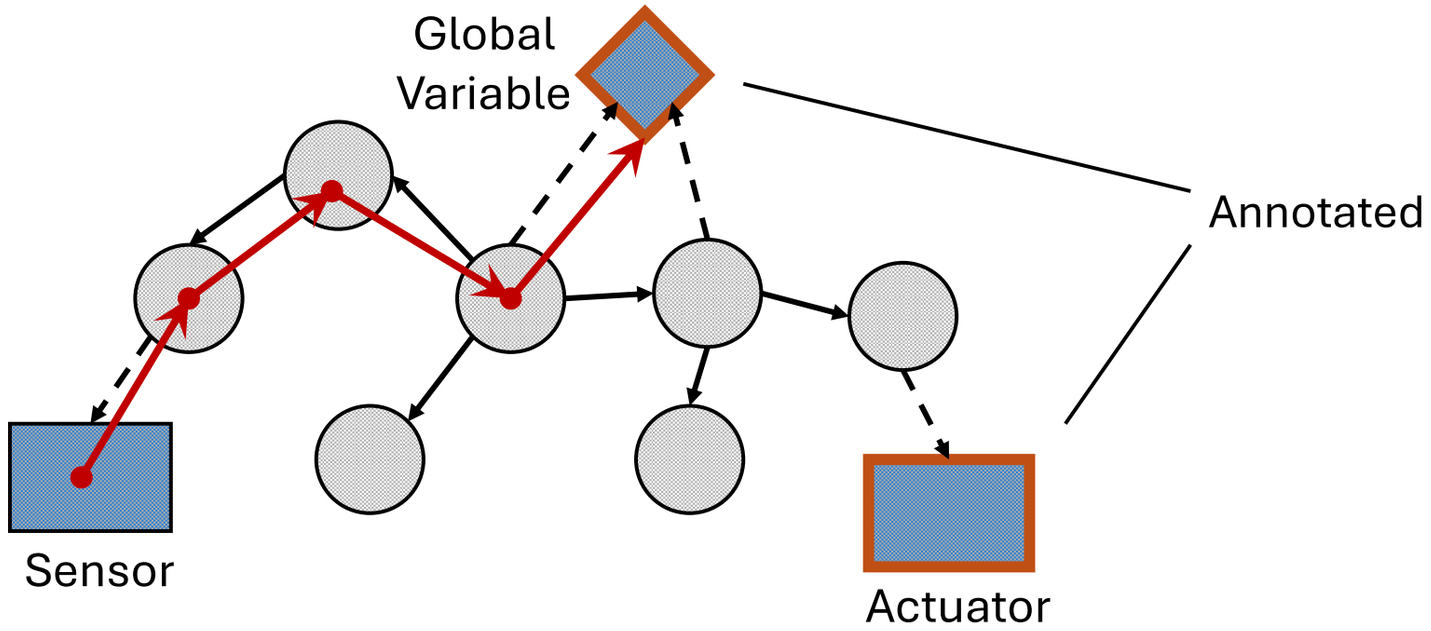
Solution 1: Sensitive Data Flow-based Compartment



Solution 1: Sensitive Data Flow-based Compartment

Backward Slicing:

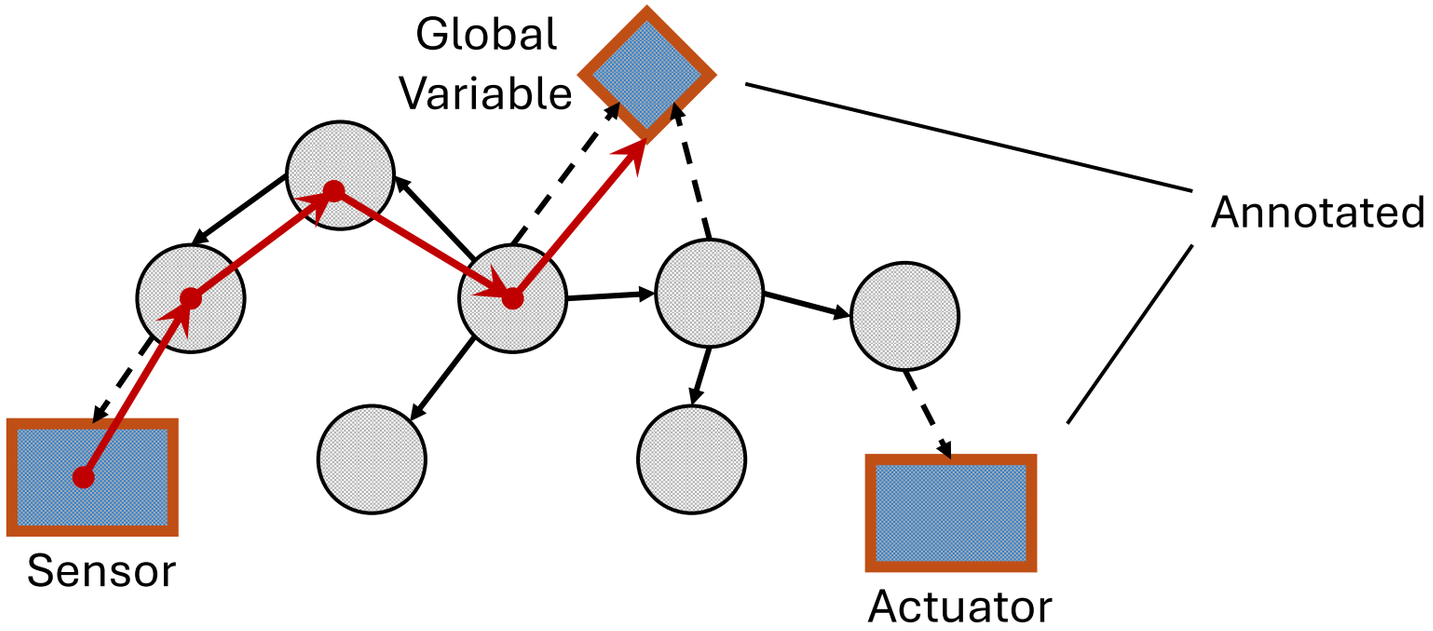
- Tracks all instructions and data objects that influence sensitive data
- Ensures integrity



Solution 1: Sensitive Data Flow-based Compartment

Backward Slicing:

- Tracks all instructions and data objects that influence sensitive data
- Ensures integrity



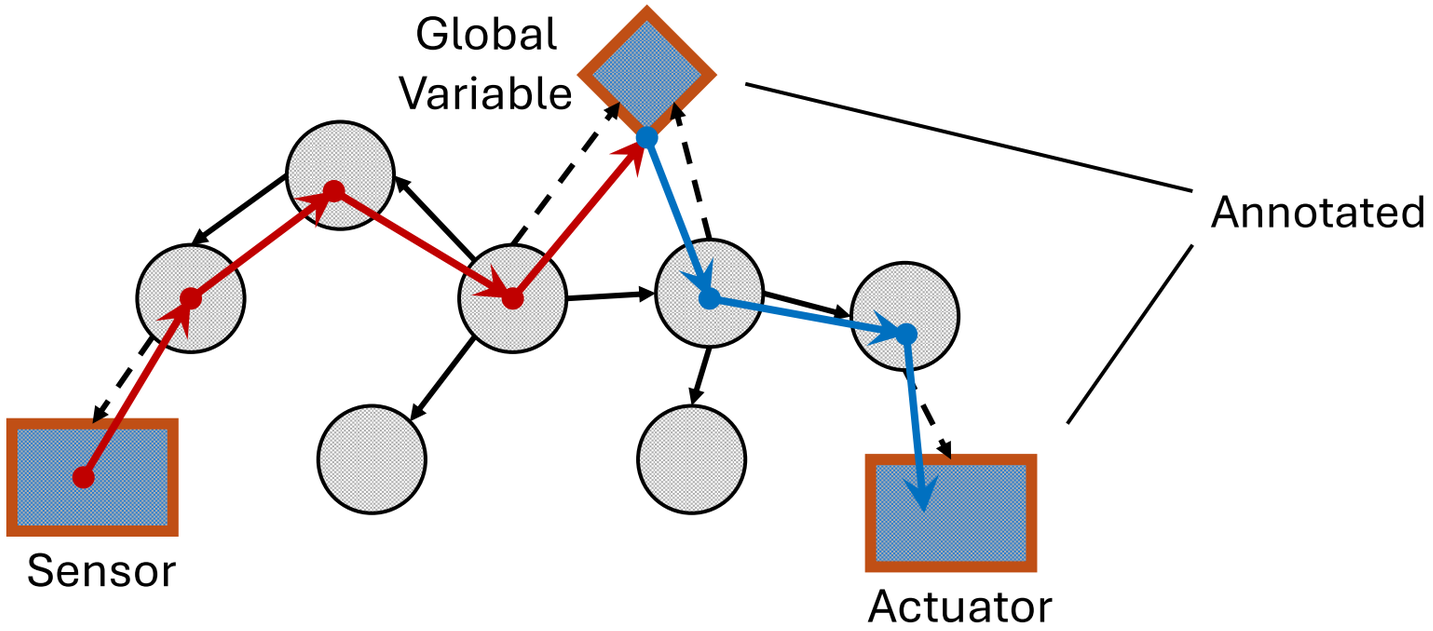
Solution 1: Sensitive Data Flow-based Compartment

Backward Slicing:

- Tracks all instructions and data objects that influence sensitive data
- Ensures integrity

Forward slicing:

- Tracks all instructions and data objects that are influenced by sensitive data
- Ensures confidentiality



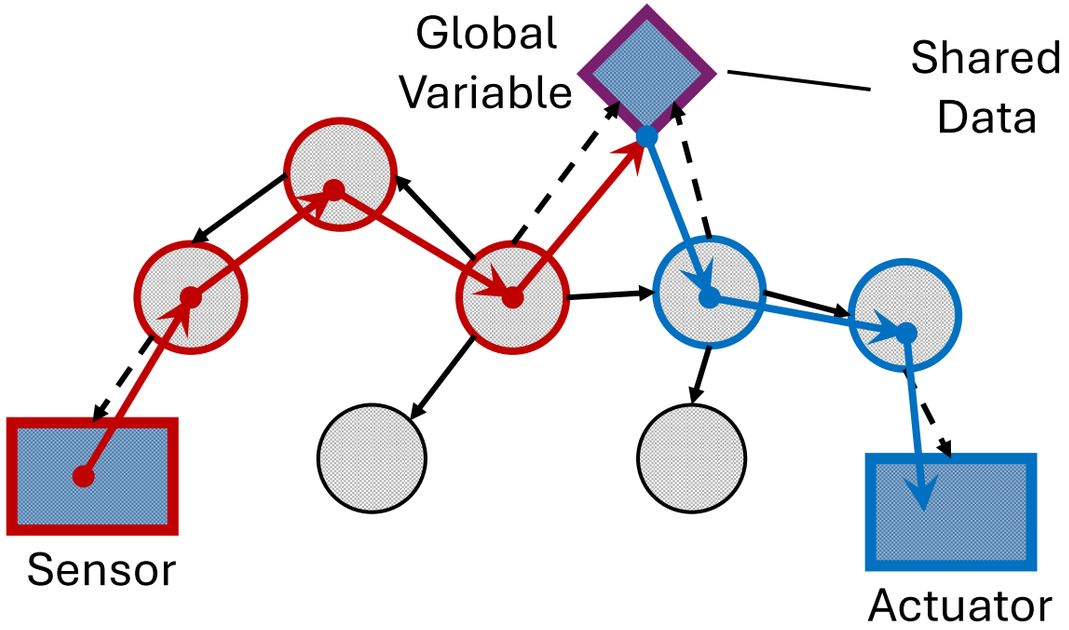
Solution 1: Sensitive Data Flow-based Compartment

Backward Slicing:

- Tracks all instructions and data objects that influence sensitive data
- Ensures integrity

Forward slicing:

- Tracks all instructions and data objects that are influenced by sensitive data
- Ensures confidentiality



Challenge 2: Lack of Intra-TEE Isolation

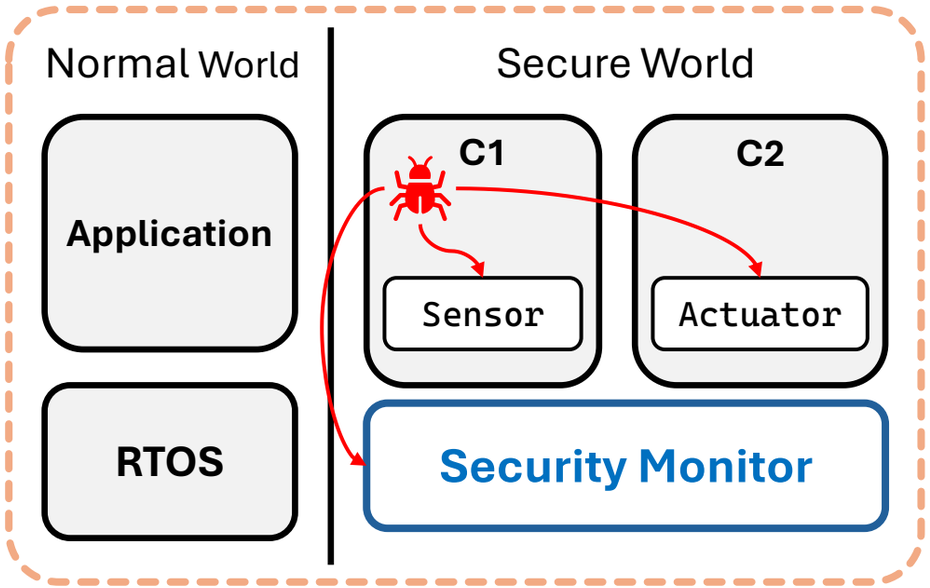
Challenge 2: Lack of Intra-TEE Isolation

One compartment can access another and even security monitor

Challenge 2: Lack of Intra-TEE Isolation

One compartment can access another and even security monitor

- Steal/manipulate sensitive data
- Bypass security checks



Solution 2: Software Fault Isolation

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer

```
int func1() {
    fp = func2; // function pointer

    var = (*fp)(arg1, arg2);
}
```

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer
- Indirect memory accesses

```
int func1() {
    fp = func2; // function pointer

    var = (*fp)(arg1, arg2);
}
```

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer
- Indirect memory accesses

```
int func1() {
    fp = func2; // function pointer

    var = (*fp)(arg1, arg2);
}
```

```
int global_var;
int func3() {
    int *ptr = &global_var;

    global_var = var;
}
```

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer
- Indirect memory accesses

Compile-time instrumentation:

- Add checks before indirect control transfer and memory accesses

```
int func1() {
    fp = func2; // function pointer

    var = (*fp)(arg1, arg2);
}
```

```
int global_var;
int func3() {
    int *ptr = &global_var;

    global_var = var;
}
```

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer
- Indirect memory accesses

Compile-time instrumentation:

- Add checks before indirect control transfer and memory accesses

```
int func1() {
    fp = func2; // function pointer
    check(fp);
    var = (*fp)(arg1, arg2);
}
```

```
int global_var;
int func3() {
    int *ptr = &global_var;

    global_var = var;
}
```

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer
- Indirect memory accesses

Compile-time instrumentation:

- Add checks before indirect control transfer and memory accesses

```
int func1() {
    fp = func2; // function pointer
    check(fp);
    var = (*fp)(arg1, arg2);
}
```

```
int global_var;
int func3() {
    int *ptr = &global_var;
    check(ptr);
    global_var = var;
}
```

Solution 2: Software Fault Isolation

Software Fault Isolation (SFI):

- Indirect control transfer
- Indirect memory accesses

Compile-time instrumentation:

- Add checks before indirect control transfer and memory accesses

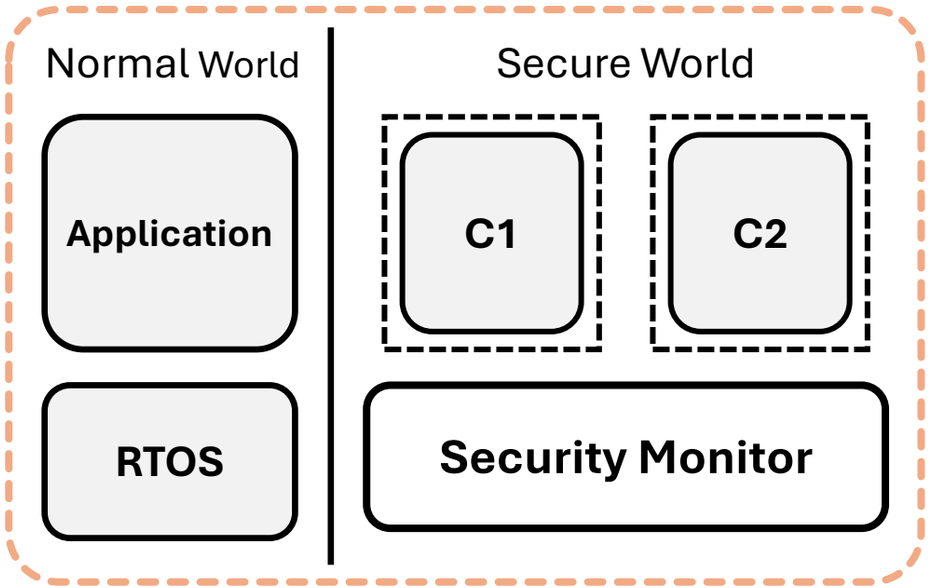
Runtime enforcement by the security monitor:

- Isolating accesses within the compartment

```
int func1() {
    fp = func2; // function pointer
    check(fp);
    var = (*fp)(arg1, arg2);
}
```

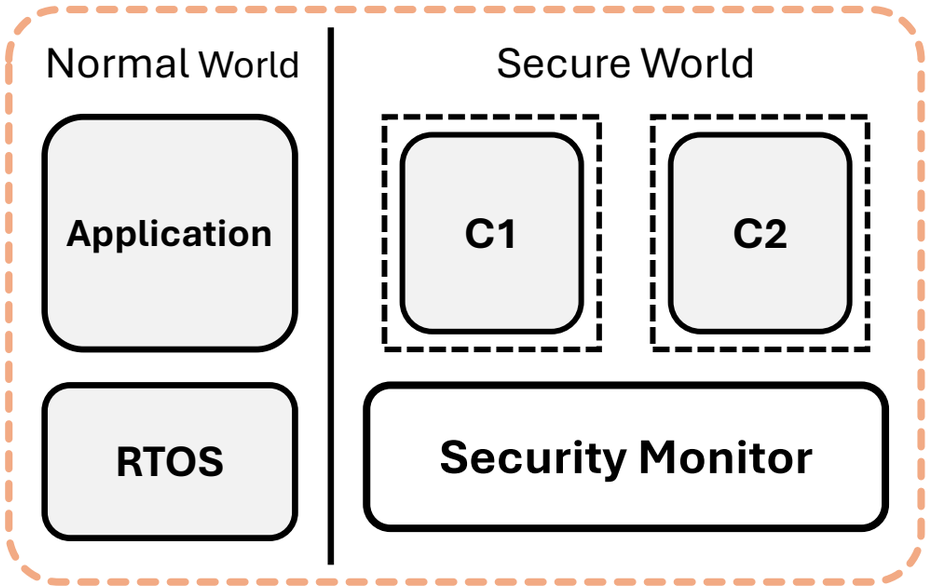
```
int global_var;
int func3() {
    int *ptr = &global_var;
    check(ptr);
    global_var = var;
}
```

Challenge 3: Shared Data/Peripheral Protection



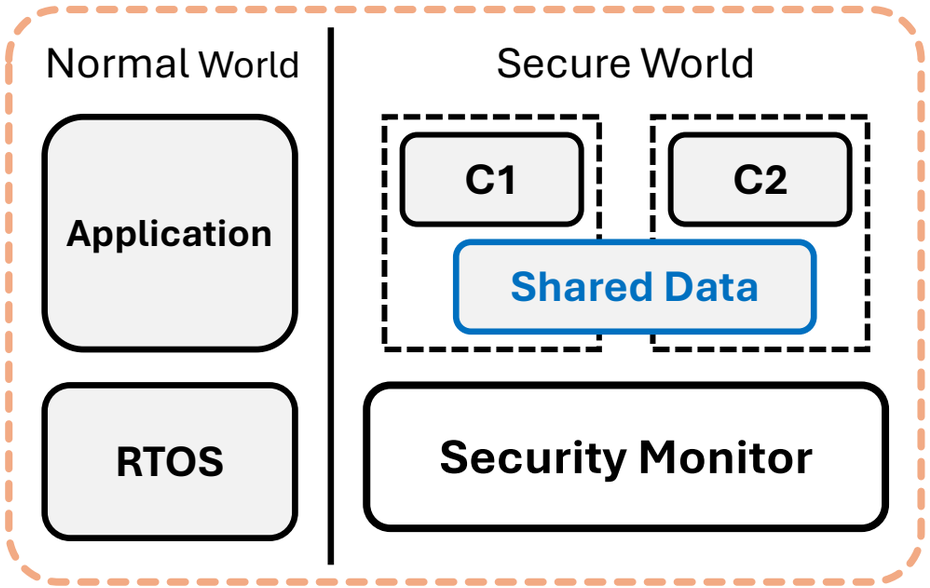
Challenge 3: Shared Data/Peripheral Protection

- Compartments may share data/peripherals



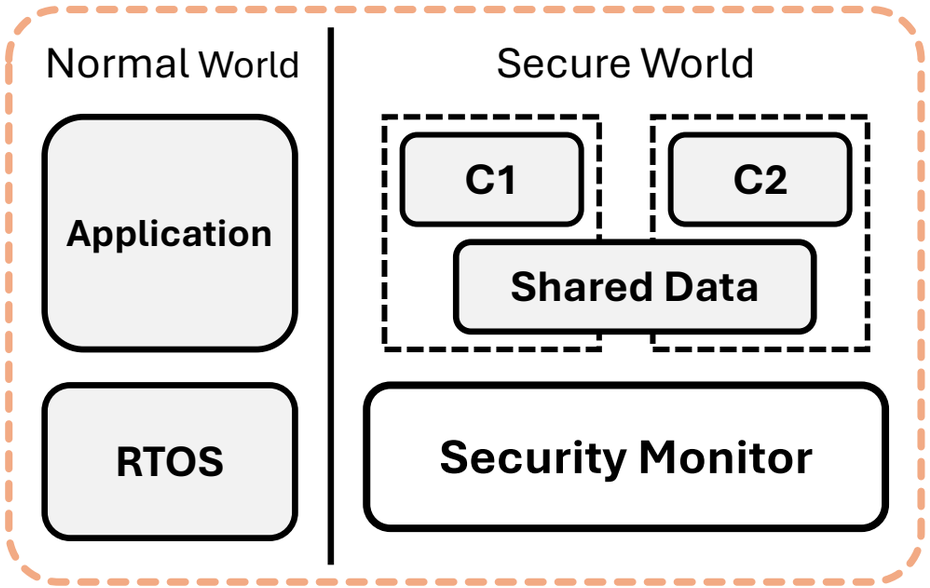
Challenge 3: Shared Data/Peripheral Protection

- Compartments may share data/peripherals



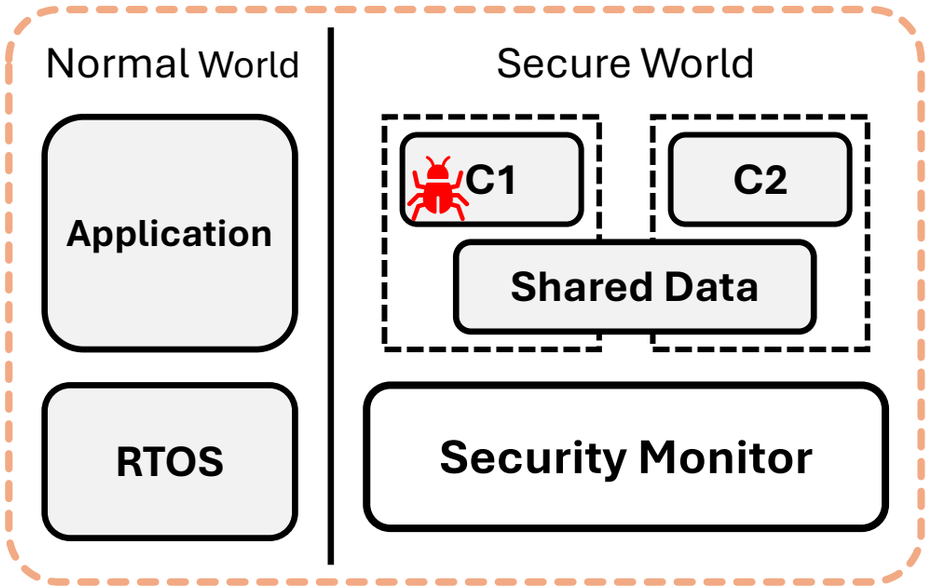
Challenge 3: Shared Data/Peripheral Protection

- Compartments may share data/peripherals
- Adversaries may exploit this to illegally access other compartments



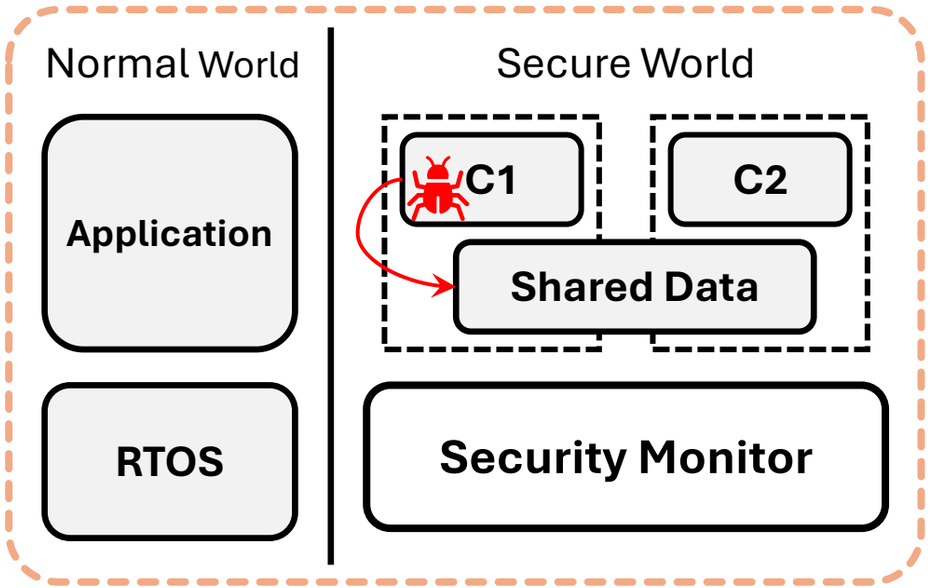
Challenge 3: Shared Data/Peripheral Protection

- Compartments may share data/peripherals
- Adversaries may exploit this to illegally access other compartments



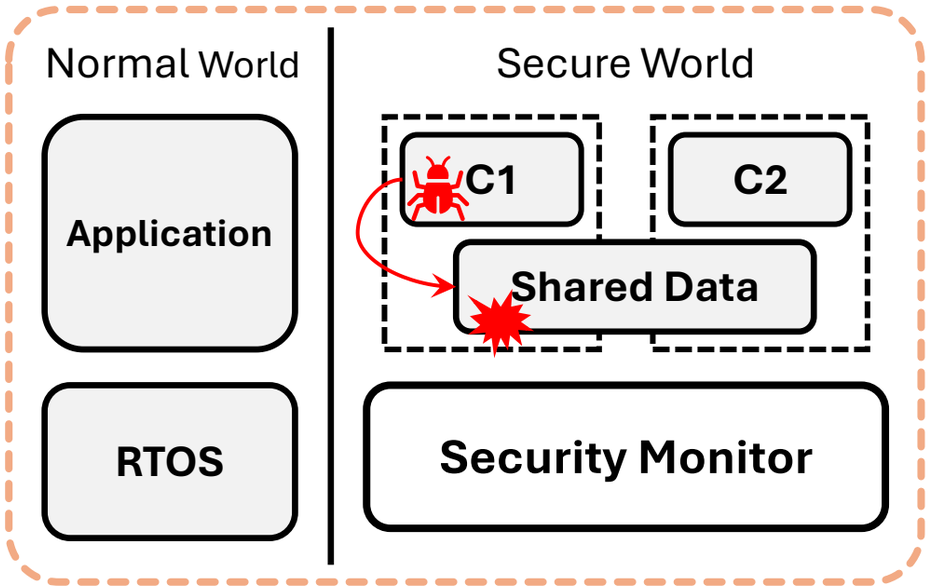
Challenge 3: Shared Data/Peripheral Protection

- Compartments may share data/peripherals
- Adversaries may exploit this to illegally access other compartments



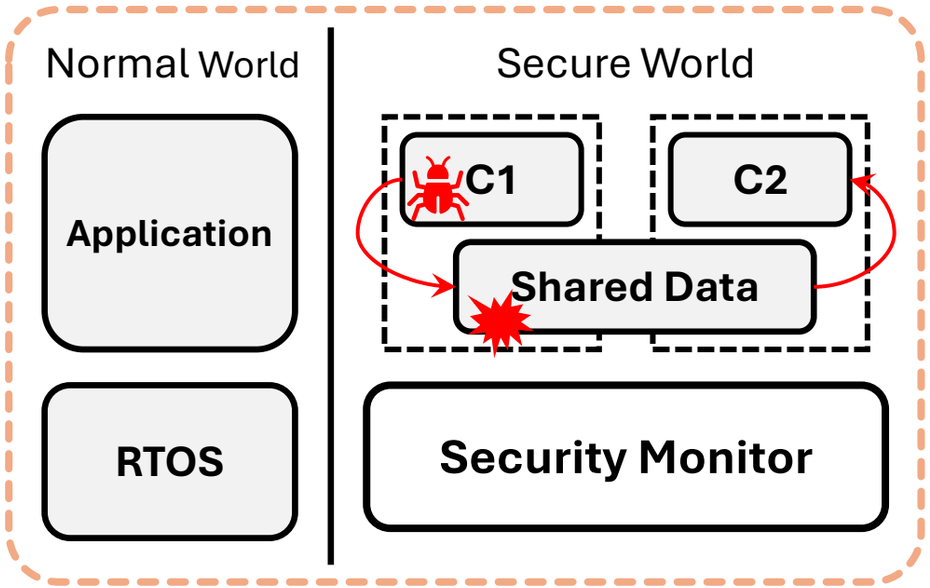
Challenge 3: Shared Data/Peripheral Protection

- Compartments may share data/peripherals
- Adversaries may exploit this to illegally access other compartments



Challenge 3: Shared Data/Peripheral Protection

- Compartments may share data/peripherals
- Adversaries may exploit this to illegally access other compartments



Solution 3: CFI/DFI for Shared Data/Peripheral

Solution 3: CFI/DFI for Shared Data/Peripheral

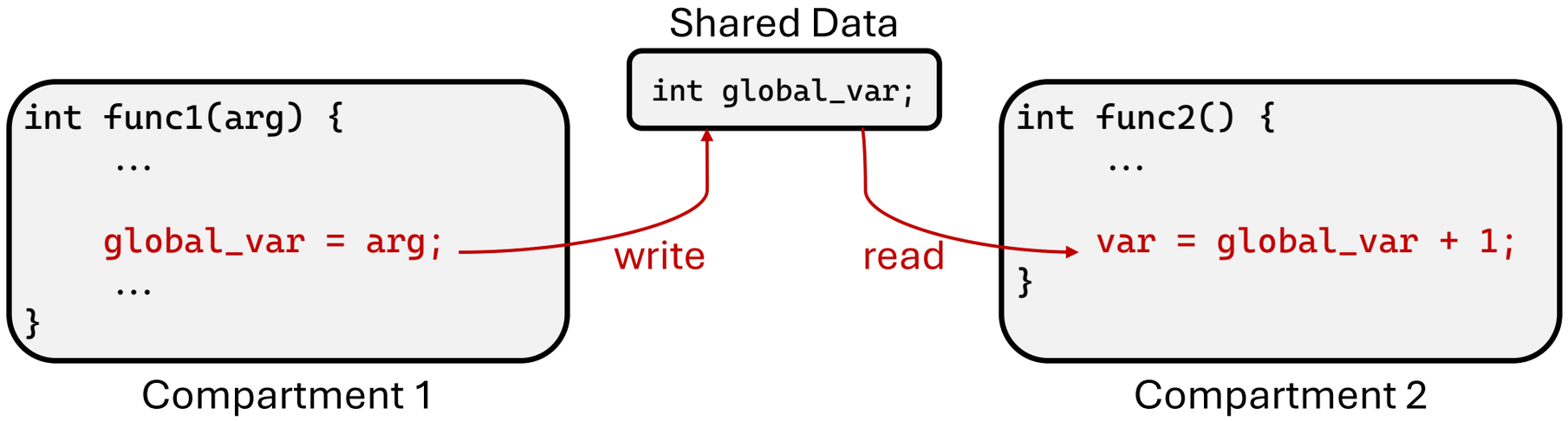
Control/Data flow integrity (CFI+DFI)

- Control/data flow leading to shared data

Solution 3: CFI/DFI for Shared Data/Peripheral

Control/Data flow integrity (CFI+DFI)

- Control/data flow leading to shared data



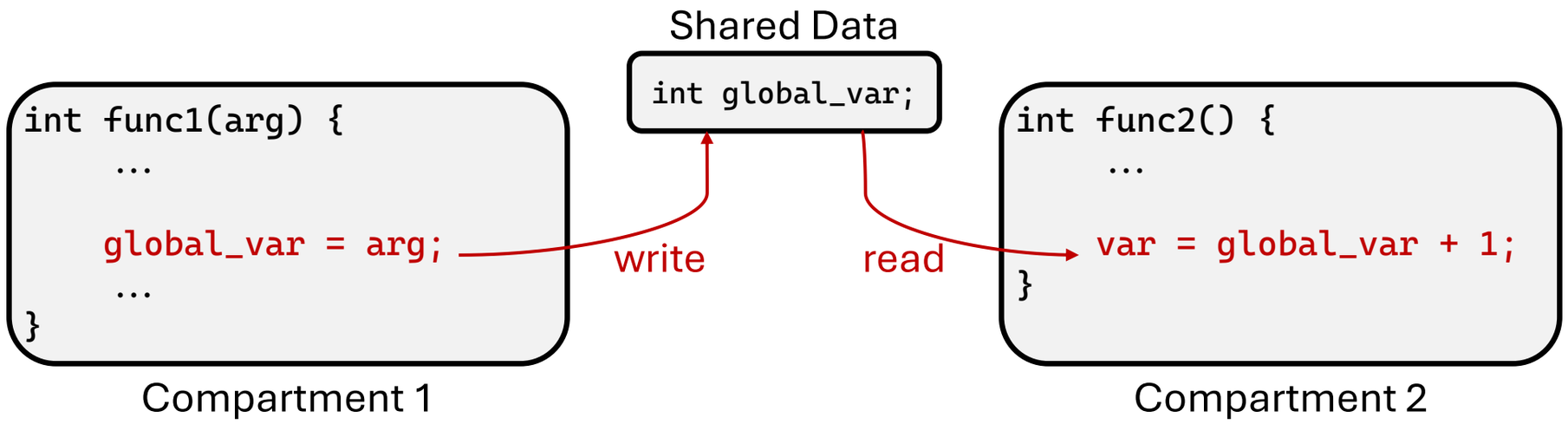
Solution 3: CFI/DFI for Shared Data/Peripheral

Control/Data flow integrity (CFI+DFI)

- Control/data flow leading to shared data

Compile-time instrumentation:

- Add checks before shared data accesses



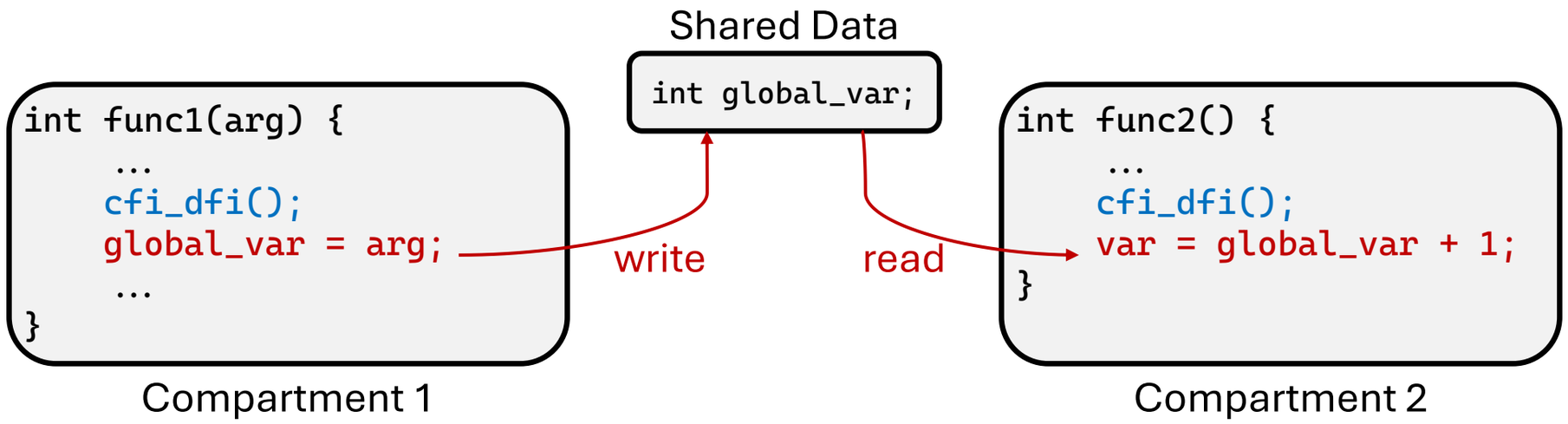
Solution 3: CFI/DFI for Shared Data/Peripheral

Control/Data flow integrity (CFI+DFI)

- Control/data flow leading to shared data

Compile-time instrumentation:

- Add checks before shared data accesses



Solution 3: CFI/DFI for Shared Data/Peripheral

Control/Data flow integrity (CFI+DFI)

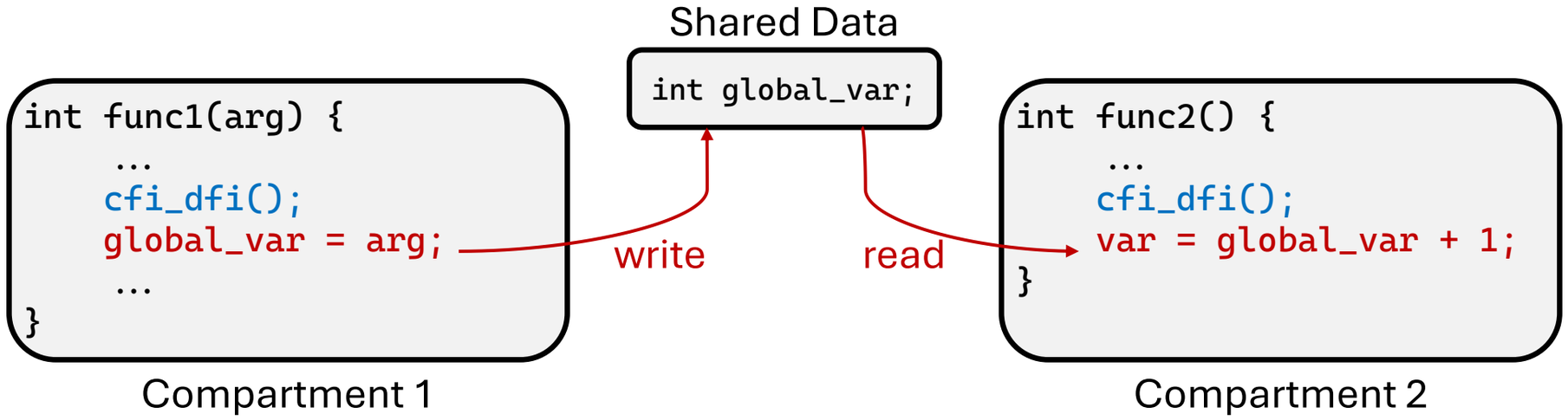
- Control/data flow leading to shared data

Compile-time instrumentation:

- Add checks before shared data accesses

Runtime enforcement by security monitor:

- When reading data, check that it came from an allowed writing

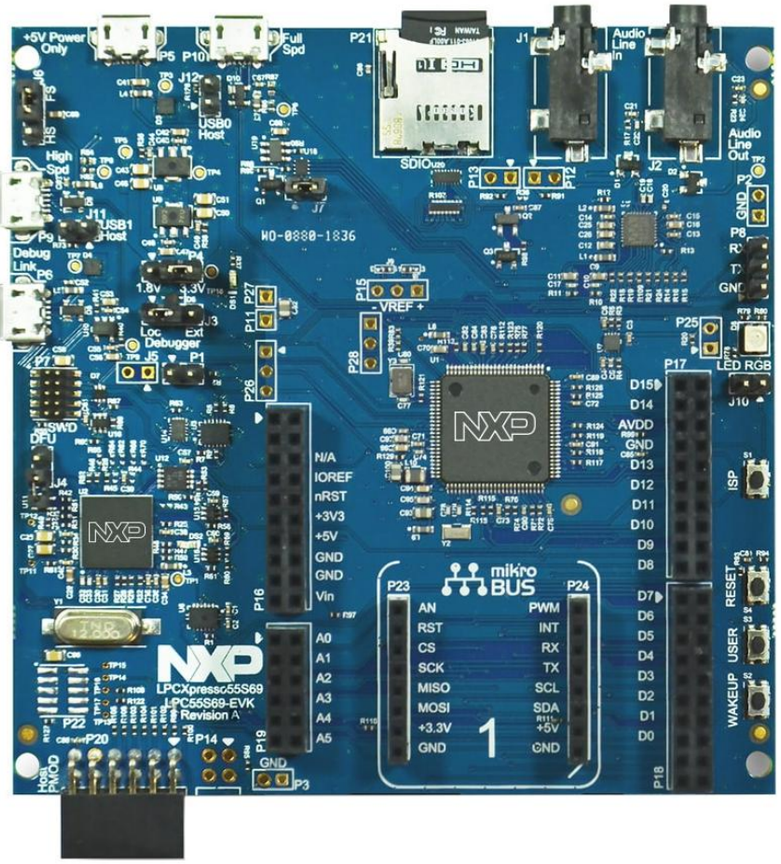


Experimental Setup

Experimental Setup

LPCXpresso55S69 development board

- ARM Cortex-M33 processor (Armv8-M)



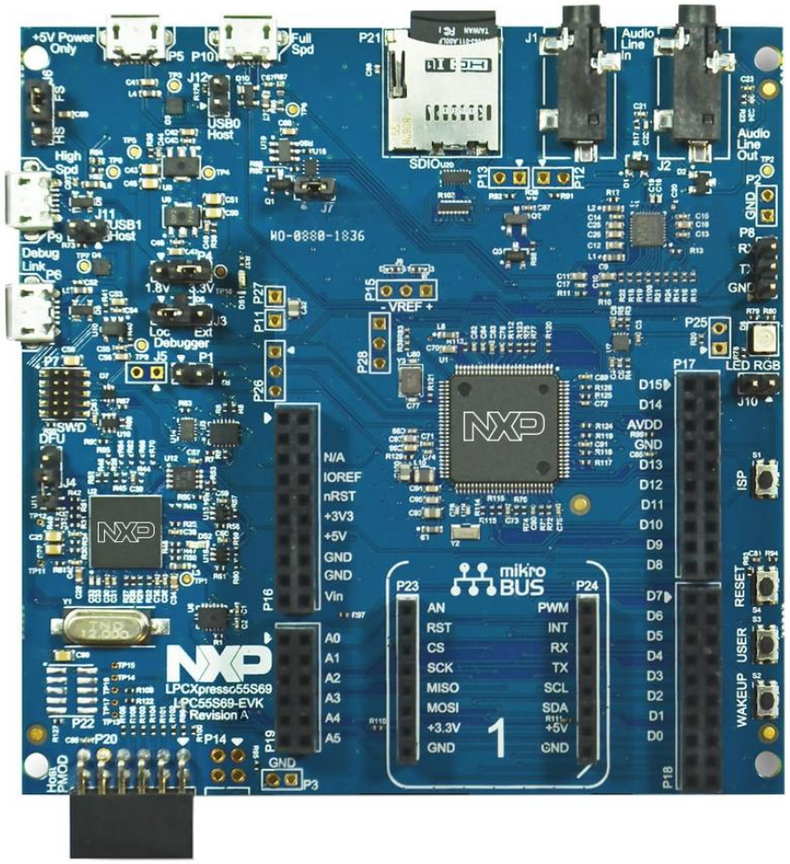
Experimental Setup

LPCXpresso55S69 development board

- ARM Cortex-M33 processor (Armv8-M)

Evaluated on 12 different bare-metal and RTOS applications:

- **Bare-metal**: PinLock, Temp, Accel, Gyro, SD-FatFS, USBVCom
- **RTOS-based**: FreeRTOS variants of the above applications



Experimental Setup

LPCXpresso55S69 development board

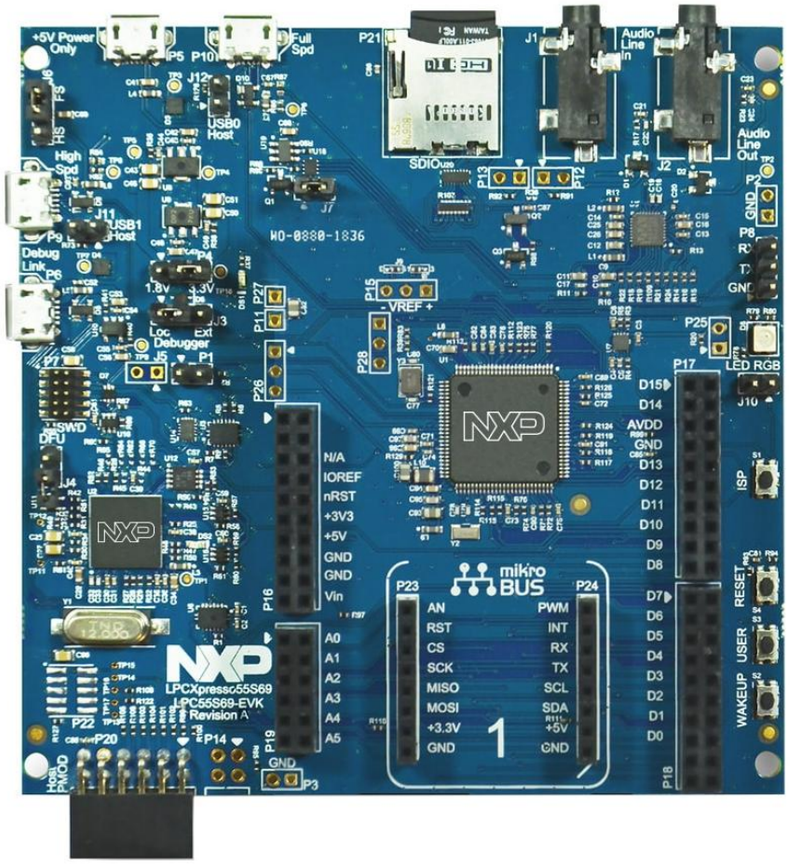
- ARM Cortex-M33 processor (Armv8-M)

Evaluated on 12 different bare-metal and RTOS applications:

- **Bare-metal**: PinLock, Temp, Accel, Gyro, SD-FatFS, USBVCom
- **RTOS-based**: FreeRTOS variants of the above applications

Other compartmentalization approach

- Thread, function, component



Security Evaluation

Security Evaluation

Baseline: No isolation

Security Evaluation

Address Space Reduction

Granularity	Compartment	Rate
	SDF	80.8%
Fine	Function	96.2%
Coase	Component	38.4%
	Thread	62.7%

Baseline: No isolation

Address Space Reduction:

Security Evaluation

Address Space Reduction

Granularity	Compartment	Rate
	SDF	80.8%
Fine	Function	96.2%
Coase	Component	38.4%
	Thread	62.7%

Baseline: No isolation

Address Space Reduction:

- Achieved average 80.8% reduction

Security Evaluation

Address Space Reduction

Granularity	Compartment	Rate
	SDF	80.8%
Fine	Function	96.2%
Coase	Component	38.4%
	Thread	62.7%

Baseline: No isolation

Address Space Reduction:

- Achieved average 80.8% reduction
- Average 30% more reduction than coarse-grained

Security Evaluation

Address Space Reduction

Granularity	Compartment	Rate
	SDF	80.8%
Fine	Function	96.2%
Coase	Component	38.4%
	Thread	62.7%

 # ROP Gadgets Reduction

Compartment	Rate
SDF	88.6%
Function	98.6%
Component	63.1%
Thread	78.5%

Baseline: No isolation

Address Space Reduction:

- Achieved average 80.8% reduction
- Average 30% more reduction than coarse-grained

ROP gadgets:

Security Evaluation

Address Space Reduction

Granularity	Compartment	Rate
	SDF	80.8%
Fine	Function	96.2%
Coase	Component	38.4%
	Thread	62.7%

 # ROP Gadgets Reduction

Compartment	Rate
SDF	88.6%
Function	98.6%
Component	63.1%
Thread	78.5%

Baseline: No isolation

Address Space Reduction:

- Achieved average 80.8% reduction
- Average 30% more reduction than coarse-grained

ROP gadgets:

- Achieved average 80.8% reduction

Security Evaluation

Address Space Reduction

Granularity	Compartment	Rate
	SDF	80.8%
Fine	Function	96.2%
Coase	Component	38.4%
	Thread	62.7%

 # ROP Gadgets Reduction

Compartment	Rate
SDF	88.6%
Function	98.6%
Component	63.1%
Thread	78.5%

Baseline: No isolation

Address Space Reduction:

- Achieved average 80.8% reduction
- Average 30% more reduction than coarse-grained

ROP gadgets:

- Achieved average 80.8% reduction
- Average 18% more reduction than coarse-grained

Performance Evaluation

Performance Evaluation

Runtime Overhead

Granularity	Compartment	Rate
	SDF	14.7%
Fine	Function	64.5%
Coase	Component	12.9%
	Thread	12.7%

Runtime overhead:

Performance Evaluation

Runtime Overhead

Granularity	Compartment	Rate
	SDF	14.7%
Fine	Function	64.5%
Coase	Component	12.9%
	Thread	12.7%

Runtime overhead:

- Incurs an average 14.7% runtime overhead

Performance Evaluation

Runtime Overhead

Granularity	Compartment	Rate
	SDF	14.7%
Fine	Function	64.5%
Coase	Component	12.9%
	Thread	12.7%

Runtime overhead:

- Incurs an average 14.7% runtime overhead
- 1.4% compartment switch; 6.3% SFI; 7.0% CFI/DFI

Performance Evaluation

Runtime Overhead

Granularity	Compartment	Rate
	SDF	14.7%
Fine	Function	64.5%
Coase	Component	12.9%
	Thread	12.7%

Memory Overhead

SDF	Overhead
Security Monitor	16.7 KB
Meta data	136 Bytes
Memory Pool	4 KB

Runtime overhead:

- Incurs an average 14.7% runtime overhead
- 1.4% compartment switch; 6.3% SFI; 7.0% CFI/DFI

Memory overhead:

- Incurs an average 31.4% memory overhead

Conclusion

Conclusion

- Use ARM TrustZone to protect against strong adversaries

Conclusion

- Use ARM TrustZone to protect against strong adversaries
- TZ-DATASHIELD:
  - Compartmentalization: Sensitive data flow
  - Intra-TEE isolation: SFI
  - Shared data/peripheral protection: CFI/DFI

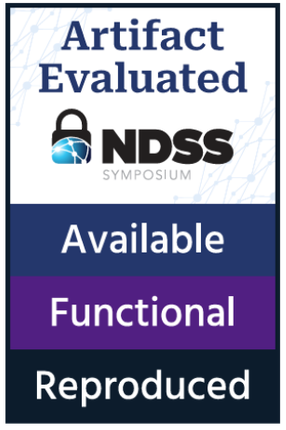
Conclusion

- Use ARM TrustZone to protect against strong adversaries
- TZ-DATASHIELD:
  - Compartmentalization: Sensitive data flow
  - Intra-TEE isolation: SFI
  - Shared data/peripheral protection: CFI/DFI
- 80.8% address space and 88.6% ROP gadget reductions

Conclusion

- Use ARM TrustZone to protect against strong adversaries
- TZ-DATASHIELD:
  - Compartmentalization: Sensitive data flow
  - Intra-TEE isolation: SFI
  - Shared data/peripheral protection: CFI/DFI
- 80.8% address space and 88.6% ROP gadget reductions
- 14.7% runtime overhead and 31.4% memory overhead

Thanks for listening.
Questions?



Code

Artifacts

Protecting IRQ Handlers

- IRQ handlers are also isolated into separate SDF compartments
- Secure Interrupt dispatcher:
  - Registered in the interrupt vector table (IVT)
  - **Intercepts IRQ requests** before invoking the actual handler

Comparison with Existing CFI/DFI

Unlike general CFI/DFI that checks universally

Selectively activates CFI/DFI only when accessing shared peripherals or data

Adjustable previous address targets

Lightweight

Annotation

```
/* Global data, confidentiality protection */
const uint8_t key_stored[KEY_SIZE] TZDS_DATA_R = {0x...};
void func() {
  /* Stack data, integrity protection */
  uint8_t buffer[BUFFER_SIZE] TZDS_DATA_W = {0x0};
  ...
}
/* Heap data, confidentiality and integrity protection */
static void *m_head TZDS_HEAP_RW = malloc(...);
/* Peripheral data at [GPIO_BASE,GPIO_BASE+0x1000),
   integrity protection */
#define GPIO_BASE (0x4008C000u)
TZDS_MMIO_W(GPIO_BASE, 0x1000)
GPIO_Type *gpio = (GPIO_Type *) GPIO_BASE;
```

Performance Overhead – CFI/DFI