Protecting Actuators in Real-Time Cyber-Physical Systems

Monowar Hasan
Assistant Professor
School of EECS, WSU
Assistant Professor
  ▶ EECS@WSU

Lab:
  ▶ Cyber-Physical Systems Security Research Lab
  ▶ https://cps2rl.github.io

Research
  ▶ Systems security

- Security for real-time and cyber-physical systems
- Trustworthy ML for embedded/IoT systems
- Resilient real-time networks using SDNs
- Security for precision agriculture
Today’s Talk

Real-Time Cyber-Physical Systems Security
Cyber-Physical Systems (CPS)

Software, Control Algorithms, Code

Networking, Communication

Microcontrollers, ECU, PLC

Sensors

Actuators

Plant
CPS Applications

- Automobiles
- Avionics
- Control Systems
- Unmanned Vehicles
- Surveillance
- Autonomous Driving
- Manufacturing
- Healthcare

* Image courtesy: Google Image Search
Real-Time CPS Security

Traditional RTS
- Custom Hardware
- Proprietary Operating System
- Proprietary Software
- Limited Network Connection

Modern RTS
- COTS Hardware
- Open Source Operating System
- Open Source Software
- More Connectivity → Internet!

Larger Attack Surface!

Modern RTS are vulnerable to security threats!

Increased Security Risks

Stuxnet Computer Worm Has Vast Repercussions

Hacker Says He Can Hijack a $35K Police Drone a Mile Away
Real-Time CPS

Distinguishing Properties
Critical Parameter: Time

Real-Time Systems (RTS) → requires both logical and temporal correctness

- Real-time ≠ fast
- Timing constraints → measured by deadlines

Example: Anti-lock Braking System (ABS)

Anti-lock Braking System (ABS) in cars
- must function correctly in milliseconds
- even 1 second delay might be too late
- car traveling at 60 mph → 88 ft. in 1 s!
Real-Time Requirement
Normal vs Late Airbag Deployment

Knocks the head back!

Normal Deployment
Late Deployment

Perturbation of timing constraints → serious consequences

Source: https://www.youtube.com/watch?v=YAwrq9-1oQQ
Real-Time CPS: Common Properties

- **Limited Resources**: computational power, energy
- **Periodic Tasks**
- **Priority-driven scheduling**

![Diagram showing time, deadline, period, job of a task, high priority, low priority, and preemption.](image)
Real-Time CPS: Common Properties

- Real-time CPS → based on **sensing** and **actuation**

false/spoofed actuation command → disrupt normal operation
How to prevent falsification of actuation commands?
Design Goals

- Examine actuation commands
  - before they are being issued to actuators

- Guarantee timing requirements

- Tamper-proof

- Better compatibility
  - design with off-the-shelf components

Trusted Execution Environments (TEE)
Model and Assumptions
System Model

- A set of fixed-priority, periodic real-time tasks
- Multicore platform, partitioned scheduling
- Each task $i$ generates $N_i$ actuation requests
Model and Assumptions

Adversary Model

- Adversarial actions result in modification of actuation commands

- No assumptions on how adversary compromises tasks
  - Known vulnerabilities
  - Remote trozans
  - Social engineering
  - ....

- No physical presence
  - Can not physically control/turn off/damage actuators
Vanilla (Non-Secure) Execution

/* regular computation */
.
.
.

/* actuation request */

set_motor_direction(DIRECTION)

...

...

UAV Controller Task

How to prevent the sending of malicious commands to actuators?
Proposed Scheme: Overview

UAV Controller Task

**Normal Execution Mode**

/* regular computation */
...
/* actuation request */

set_motor_direction(DIRECTION)

**Trusted Execution Mode**

if DIRECTION is CLOCKWISE:
/* allow */

Switch to secure mode

set direction if valid
- Designer-provided, design-time "rules" → required for correct system operations
  
  ▸ maps state → action

If MOTOR is FRONT_LEFT:
DIRECTION = CLOCKWISE

If MOTOR is FRONT_RIGHT:
DIRECTION = ANTICLOCKWISE

....
But...

we can’t always check what we wanted!

Real-time constraints!
Timing constraints?

Execution Time + Checking Overhead + Preemption ≤ Deadline

NOT insignificant for TEEs!
Background - ARM TrustZone

- ARM TrustZone isolates trusted software and data.

Sequence of steps:

1. SMC
2. Normal World
3. Secure Monitor
4. Secure World
5. Trusted Kernel (OP-TEE)
6. Trusted User Applications
7. Trusted Kernel (OP-TEE)
8. Untrusted

This diagram illustrates the separation between trusted and untrusted applications and the secure monitor (SMC) that enforces access control.
Normal to secure mode context switch is costly!

**Mode switching steps:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Function</th>
<th>Overhead (microsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Initialize</td>
<td>TEEC.InitializeContext()</td>
<td>64</td>
</tr>
<tr>
<td>2.</td>
<td>Open session</td>
<td>TEEC_OpenSession()</td>
<td>49233</td>
</tr>
<tr>
<td>3.</td>
<td>Transfer</td>
<td>TEEC_InvokeCommand()</td>
<td>146</td>
</tr>
<tr>
<td>4.</td>
<td>Close session</td>
<td>TEEC_CloseSession()</td>
<td>15682</td>
</tr>
<tr>
<td>5.</td>
<td>Clean up</td>
<td>TEEC_FinalizeContext()</td>
<td>29</td>
</tr>
</tbody>
</table>

**Overhead (microsec):**

- TEEC.InitializeContext(): 64 microsec
- TEEC_OpenSession(): 49233 microsec
- TEEC_InvokeCommand(): 146 microsec
- TEEC_CloseSession(): 15682 microsec
- TEEC_FinalizeContext(): 29 microsec

**Total Overhead:** 66 ms

*Ref: rover/drone operates at 5 Hz (200 ms)*
How to reduce checking overheads?

- Check a subset of actuation commands

Which subset?
Our Approach

SCATE

Selective Checking And Trusted Execution

SCATE: Key Idea

- For each job of a task:
  - non-deterministically select a subset of commands for checking

SCATE checks few commands

But...

From adversary’s view:

SCATE checks it all!
**SCATE Example**

- **Let:**
  - A task generates 3 actuation commands $a_1, a_2, a_3$
  - We can only check 2 commands

- **Goal:**
  - Check **all** commands $\rightarrow$ adversary's view

- **Runtime random selection:**
  - Let $p_1 = 0.5$, $p_2 = 0.4$, $p_3 = 0.1$

- **Fitness Proportionate Selection**
  - Generate random number $R$ in $[0, 1]$
    - If $R$ in $[0, 0.5) \rightarrow$ check $(a_1, a_2)$
    - If $R$ in $[0.5, 0.9) \rightarrow$ check $(a_2, a_3)$
    - If $R$ in $[0.9, 1.0]$ \rightarrow check $(a_1, a_3)$
Let:
- A task generates 3 actuation commands $a_1, a_2, a_3$
- We can only check 2 commands

Goal:
- Check all commands $\rightarrow$ adversary’s view

Three possible choices:
- $(a_1, a_2)$
- $(a_2, a_3)$
- $(a_1, a_3)$

Assign probabilities:
- $p_1$
- $p_2$
- $p_3$

How to derive these probabilities?

- Game theoretical analysis
  - Two-player game
  - Linear optimization problem
**SCATE**
*Game Theoretical Analysis*

- **Two-player game**
  - **Designer** and **Attacker**

- **Parameters:**
  - **System Reward**
  - **System Cost**

- **Reward**$_{ij}$
- **Cost**$_{ij}$

- Δ *reward & † cost → GOOD for designer, BAD for attacker*
- † *reward & Δ cost → BAD for designer, GOOD for attacker*

**Formulate linear optimization problem**

- maximize reward, minimize cost
  → probabilities of checking commands
Evaluation

- **Implementation:**
  - Raspberry Pi 3
  - Linux 4.16.56
  - OP-TEE 3.4

- **RT-IoT Platforms:**
  - COTS hardware

SCATE Implementation:
https://github.com/mnwrhsn/scate_implementation
Evaluation

- Implementation:
  - Raspberry Pi 3
  - Linux 4.16.56
  - OP-TEE 3.4

- RT-IoT Platforms:
  - COTS hardware

Evaluation

SCATE Implementation:
https://github.com/mnwrhsn/scate_implementation
Evaluation
Actuation Commands, Checks Inside Enclave, and Attacks

**Actuation Commands**
- Set the speed of the wheels
- Set direction of the wheels (forward, backward, left, right)

**Security Checks**
- Speed of the motors should be within predefined range (70-100)

**Attacks**
- DoS attack → sets arbitrary high speed to the wheel motors
- Destabilize rover
Results
Impact on Detection Time

Benchmark scheme:

"Fine-grained" checking → checks all four commands

Metric: time-to-detect attacks
→ how many additional task instances SCATE requires when compared to Fine-grained checking

SCATE does not delay significantly
* Mean delay: 1 additional instances
* 99-th percentile delay: 3 additional instances
Results
Impact on Execution Time

Benchmark schemes:
- "Fine-grained" → checks all four commands
- "Unsecured" → vanilla execution (no checks)

SCATE finishes execution before deadline
Fine-grained checking misses deadline!
Experiments | Key Findings

● SCATE → on average requires 1-3 **additional instances** to detect attacks
● Fine-grained checking **does not** comply with timing requirements
● SCATE manages to complete **before** deadlines
Remarks

- SCATE:
  - Prevents falsification of actuation commands
    - Trusted Execution Environments → ARM TrustZone
  - Complies with real-time requirements → “selective checking”

✓ Implemented and evaluated on four COTS platforms
Thanks!

Questions?

https://monowarhasan.info/

https://cps2rl.github.io
Supplementary Slides
Discussion #1

How to check *all* commands but minimize TEE overheads?

- Group multiple commands and check together
  - SCATE *(partially)* supports this
- Additional evaluation:
  - When we can or can’t use this feature
SCATE causes some delay in detection →

- System may not be in “unsafe” state immediately due to inertia
- Demonstrate by experiments
Discussion #3

- 66 ms TEE overhead → platform specific
  - Raspberry Pi 3 → older, slower processor (700 MHz)
  - OP-TEE follows GlobalPlatform API standards
  - Similar findings in prior work*

* Amacher et al, On the Performance of ARM TrustZone, DAIS 2019

On The Performance of ARM TrustZone*
(PRACTICAL EXPERIENCE REPORT)

Julien Amacher and Valerio Schiavoni

Université de Neuchâtel, Switzerland, first.last@unine.ch

Fig. 7: Basic TA operations: loading, unloading and successive calls to load/unload the same TA.
Let:
- A task generates 3 actuation commands
- We can only check 2 commands

```c
/* other computation */
...
actuation_request_1()
...
actuation_request_2()
actuation_request_3()
/* other computation */
...```

Task Instance 1
Evaluation Platforms

Summary

Flight Controller
- **Actuation Commands**
  - Set PWM frequency
  - Set PWM pulse durations wheels (for four motors)
- **Security Checks**
  - Check PID control coefficients
- **Attacks**
  - Set incorrect PWM pulse values
  - Destabilize quadcopter

Robot Arm
- **Actuation Commands**
  - Set rotation angle of the arm
  - Four PWM pulses
- **Security Checks**
  - Check robot arm can move up to certain angle
- **Attacks**
  - Synchronization attack → sets incorrect angle value
  - Destabilize robot operation

Syringe Pump
- **Actuation Commands**
  - Set motor rotation frequency
  - Push/pull motors (3 each)
- **Security Checks**
  - Check number of push/pull operations
- **Attacks**
  - Injects more fluid than desired
  - Health/safety concern
Results
Impact on Detection Time

Benchmark scheme:
“Fine-grained” checking → checks all commands

Metric: time-to-detect attacks → how many additional task instances SCATE requires when compared to Fine-grained checking
Results
Impact on Execution Time

Platform: Ground Rover

Execution Time (ms)

- Unsecured: 2 ms
- Fine-grain: 261 ms
- SCATE: 132 ms

Platform: Flight Controller

Execution Time (ms)

- Unsecured: 1 ms
- Fine-grain: 325 ms
- SCATE: 131 ms

Platform: Robotic Arm

Execution Time (ms)

- Unsecured: 44 ms
- Fine-grain: 303 ms
- SCATE: 174 ms

Platform: Syringe Pump

Execution Time (ms)

- Unsecured: 3 ms
- Fine-grain: 515 ms
- SCATE: 223 ms
Results (contd.)

Impact on Detection Time

<table>
<thead>
<tr>
<th>Platform</th>
<th>Delay</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>99th percentile</td>
</tr>
<tr>
<td>Flight Controller</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Robot Arm</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Syringe Pump</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

Mean delay is no more than 3 instances

Impact on Detection Time

<table>
<thead>
<tr>
<th>Platform</th>
<th>Deadline (ms)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SCATE</td>
<td>Fine-Grain</td>
</tr>
<tr>
<td>Flight Controller</td>
<td>200</td>
<td>131</td>
</tr>
<tr>
<td>Robot Arm</td>
<td>250</td>
<td>174</td>
</tr>
<tr>
<td>Syringe Pump</td>
<td>300</td>
<td>223</td>
</tr>
</tbody>
</table>

SCATE complies with timing requirements
Fine-grained checking misses deadlines!
Limitations and Discussion

- Some attacks may not be detectable
  - Zero-day attacks?

- SCATE blocks malicious commands
  - Other possibilities → send buffered commands, raise alarms

```c
/* actuation request */
set_motor_direction(DIRECTION)

if DIRECTION is OKAY:
  /* allow */
else:
  /* deny */
```

Switch to secure mode