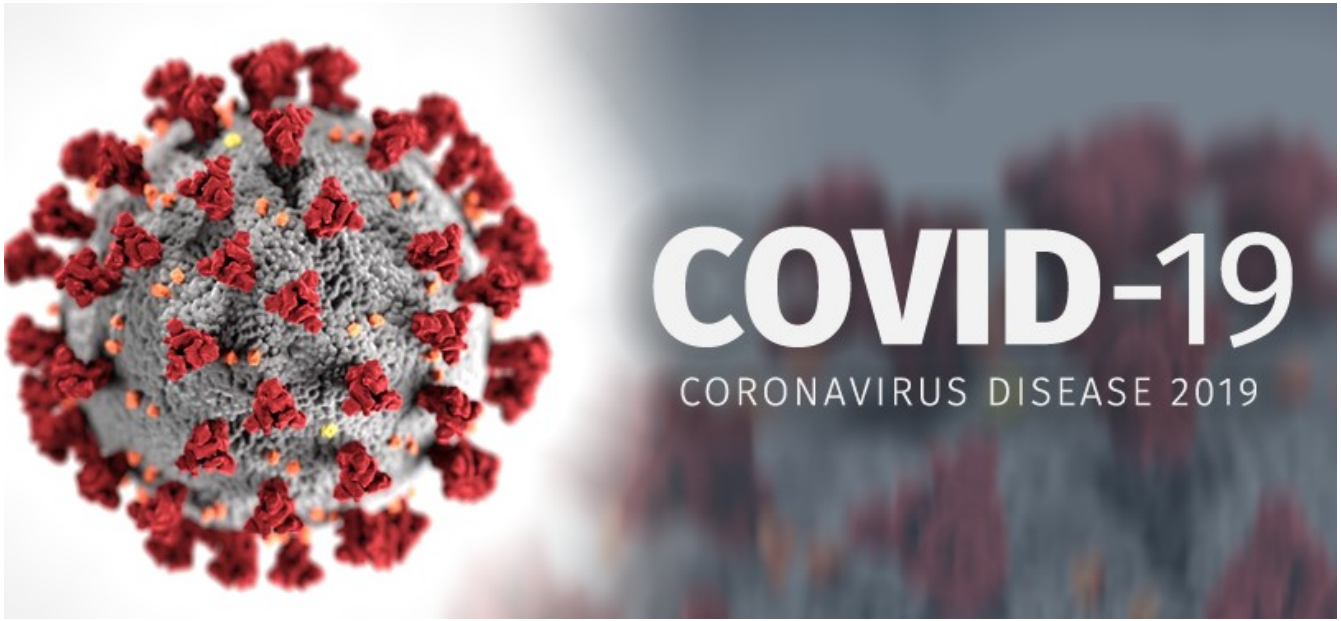
Preserving Physical Safety Under Cyber Attacks

Written by Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo

Presented by Nicholas G. Reveliotis

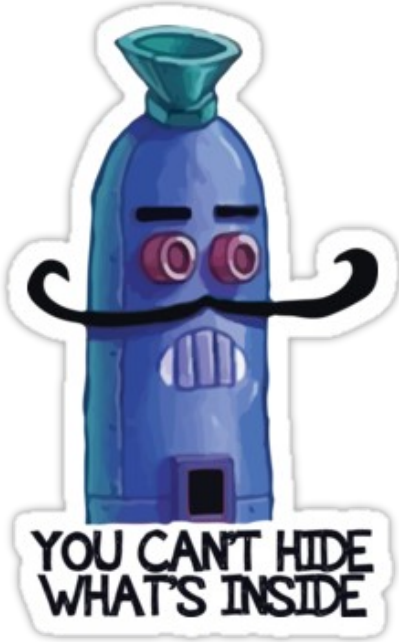
Coronavirus (COVID-19) Update

- **Due to Coronovirus (COVID19) all TCP applications are being converted to UDP to avoid handshakes**



Our Current Position

- **Major growth in Cyber-Physical Systems (CPSs) ongoing and projected to continue**
- **Much higher risks with CPSs**
  - Robot takeover anyone?
- **Requirements to protect these systems are stringent, but attacks still happen**
- **Many attacks on CPSs aim to disrupt the traditional operation of the CPS**



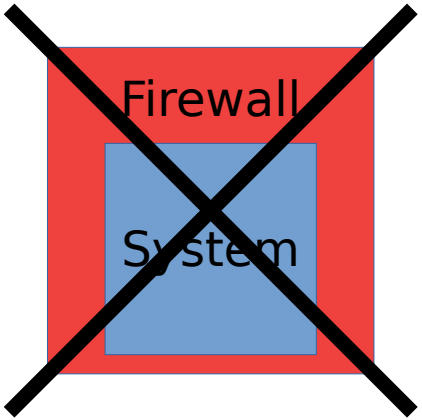
Examples

- **Stuxnet**
  - Computer worm that disrupted the centrifuges of Iranian nuclear plants
- **Medical Pacemakers**
  - Malware aimed to maliciously emit potentially fatal shocks
- **Vehicular Controllers**
  - Jeep Cherokee was hacked to allow remote startup and manipulation to the brakes, steering, and entertainment system

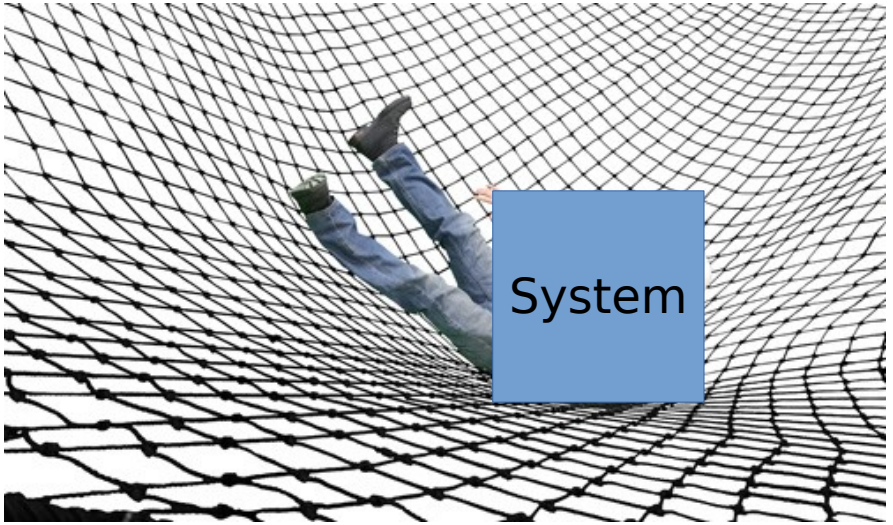
How can we protect the growing amount of CPSs with increasing risks?

A New Approach

- **What if we could stop the attacks before they are able to drastically impact the CPS?**



Not a firewall, but rather a safety net



A New Approach

- **We can’t create a bulletproof system, it will be compromised**
- **Altering a physical system takes time, there’s inertia.**
  - A plane at 25,000 feet can’t crash within seconds
  - A greenhouse can’t turn into the Sahara within seconds
- **Unlike traditional software attacks which act instantaneously, there is a period of time where failure can be prevented with CPSs**

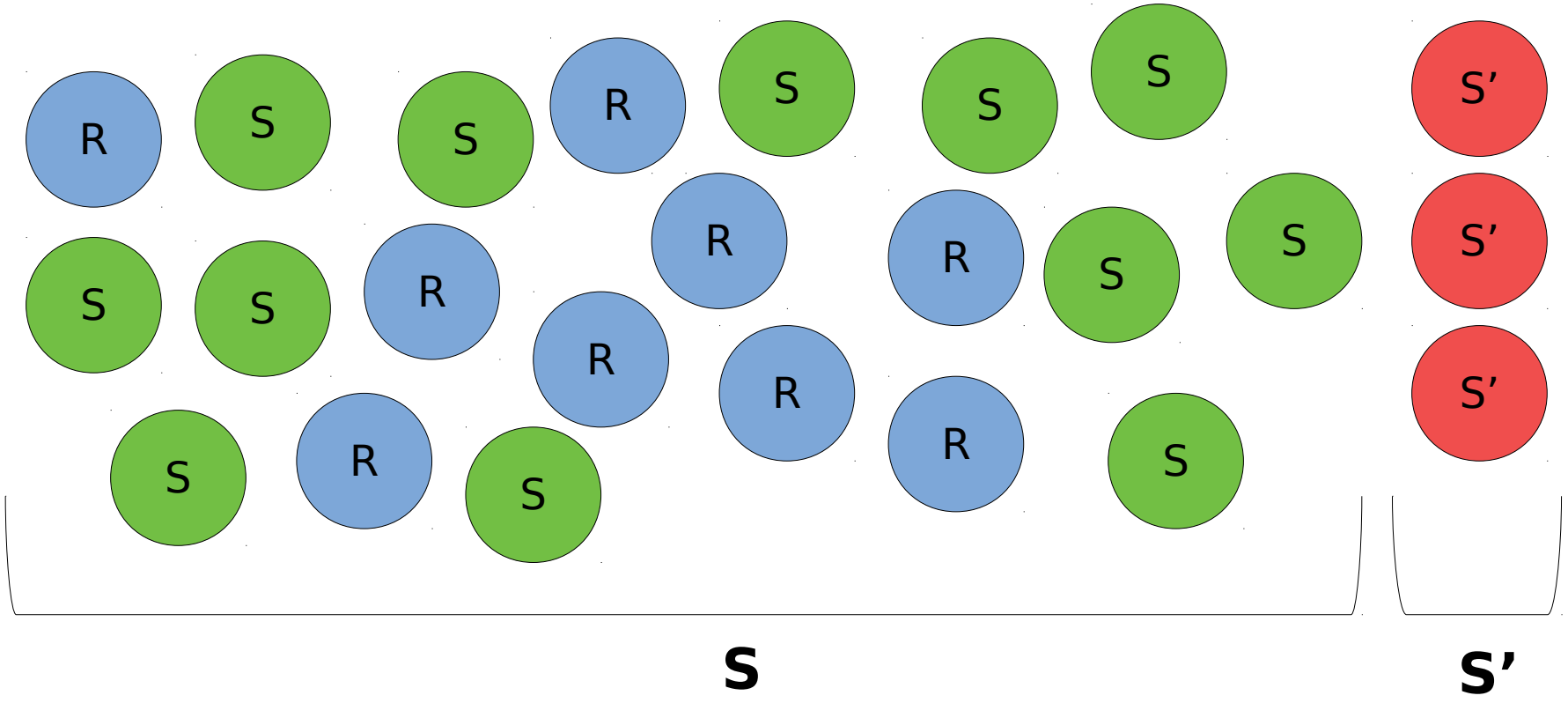
Defining our Safe Time-frame

- **There is a period of time where a drastic action can be prevented, but how long?**
- **Secure Execution Intervals (SEIs) and safety windows**
  - Dynamically calculated in realtime based upon the CPS current and possible states (more on next slide)
    - Ex: Plane at 25,000 feet vs 1,000 feet
- **The goal is: “that an attacker with full control will not have enough time to destabilize or crash the physical plant in between two consecutive intervals.”**

Understanding SEI's Calculation Algorithm

- **CPSs operate with a set of states divided into the following categories**
  - Admissible (Valid) States = S
    - States that do not violate any of the operational constraints of the physical plant
  - Inadmissible (Invalid) States = S'
  - Recoverable States = R
    - States that can be recovered from a safety controller (SC)
      - More on this later
    - R ⊆ S

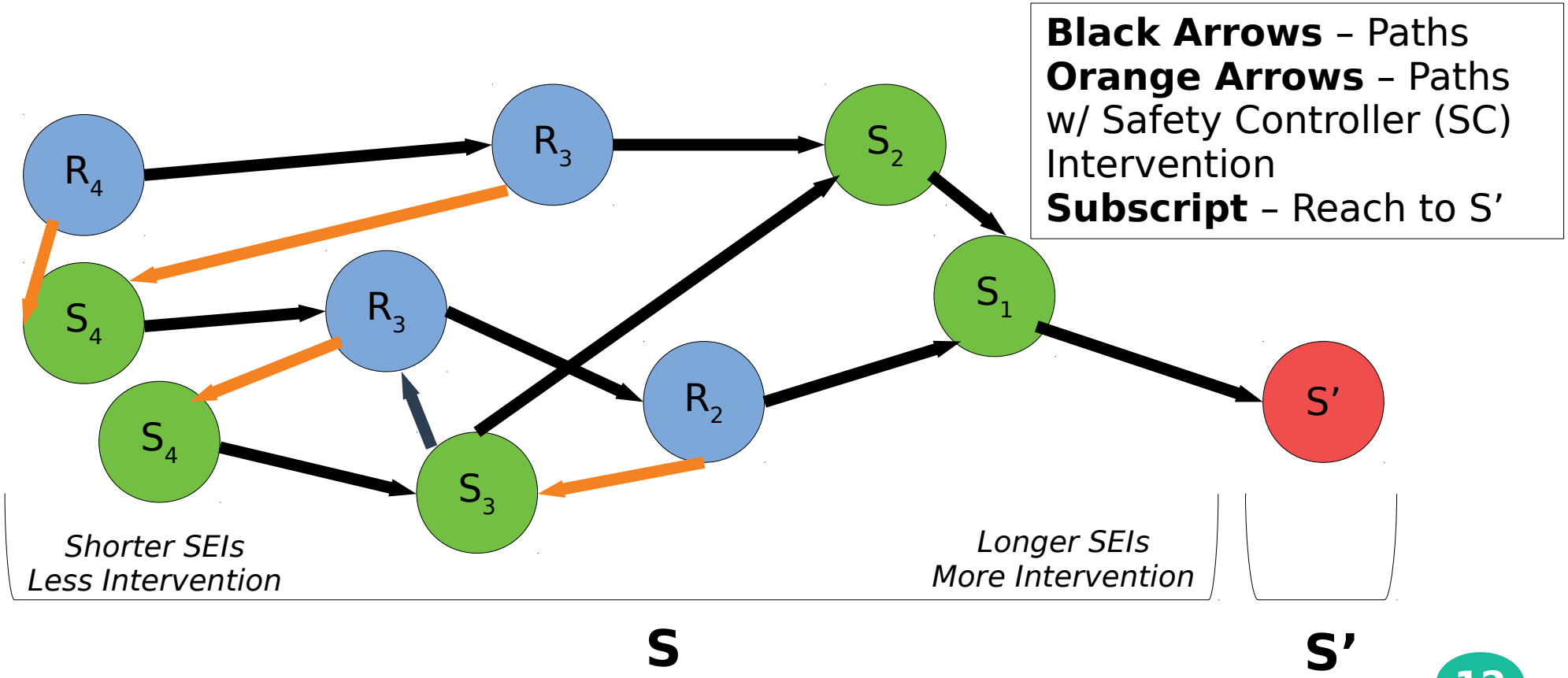
Understanding SEI's Calculation Algorithm



Understanding SEI's Calculation Algorithm

- **These states within a CPS are mapped into a multi-dimensional box and grouped into neighborhoods**
  - This links each state with all possible reachable states
- **We compute multiple reach algorithms that determine the time to reach particular states**
  - Ex: Reach ≤ T (x, C) represents all of the reachable states in a given time interval, *t*, from state *x*
- **With this data, we compute length of our SEI and our safety window (time until next SEI)**
  - If we're dangerously close to an inadmissable state (S'), we would have a shorter safety window and a longer SEI as more time would be needed to recover the CPS
  - The goal is to never reach an inadmissable state (S'), and to remain within a recoverable state (R)

Understanding SEI's Calculation Algorithm



Requirements for Calculating SEIs

- **Compromised or not, we perform our SEI calculaton**
- **Must be computed in a trusted execution environment (TEE)**
  - Our weakest-link
  - Two approaches
    1. Restart-based implementation which utilizes full system restarts and software reloads from ROM
    2. TEE-based implementation which utilizes TEE such as ARM TrustZone [43] or Intel’s Trusted Execution Technology (TXT) [24] that are available in some hardware platforms

Requirements for Calculating SEIs

- **Compromised or not, we perform our SEI calculaton**
- **Mu**
  env
  - C
  - T

Question by Lily Shpak:

After an attack the system needs time to stabilize the plant, does the time it takes to stabilize the system give the attacker another opportunity to attack?

rts and

ustZone
e

available in some hardware platforms

Overall Procedure

- **Regardless of the implementation utilized, the same procedure is followed**
  - Retrieve current state and determine range to inadmissable state/system failure
  - Use the safety controller (SC) to move the CPS further from failure
  - Perform SEI calculation (within a TEE from one of these implementations) to ensure no compromise can severely damage the CPS
  - System resumes operation
  - Hit our next SEI
  - Repeat

Let’s Play a Game

Restart-based Implementation

- **Forces full reboots that pull OS from secure ROM**
- **Requires designated hardware Root of Trust (RoT) module to manage rebooting the system at a frequency that can only be set once**
  - Maintaining this isolation is a necessity to prevent attacks from modifying SEI calculation. Once it’s timer is set, the RoT device is disconnected from the system, and only pings the device’s restart pin once it’s timer expires.
- **Compromised systems are no longer compromised post-reboot**

Restart-based Implementation

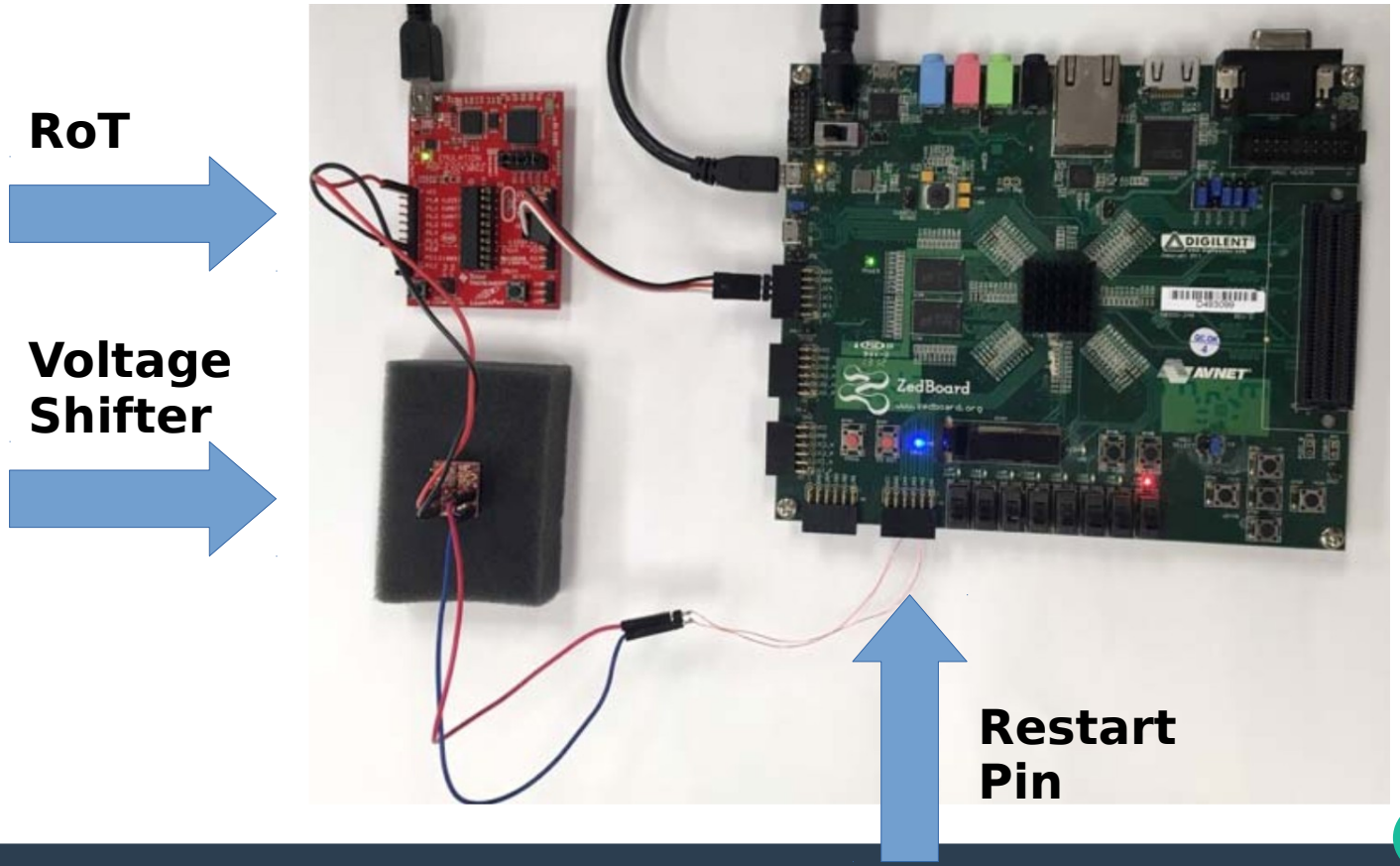
- **Forces full reboots that pull OS from secure ROM**
- **Rec… …RoT) mo… free**
  - M… …ks from n… …device is… …vice's r…
- **Cor… …ised post-reboot**

Question by Graham Schock:

The paper makes the assumption that the ROT, the separate hardware module that periodically sends restart signals can not be hacked. If we make the assumption that the whole device can not be hacked, how sure are we of the security of this module.

Restart-based Implementation's RoT Module

The RoT allows for the restart timer to be set only **once**. Once that timer expires, the restart pin of the system is hit with a voltage, rebooting the system.



Restart-based Example

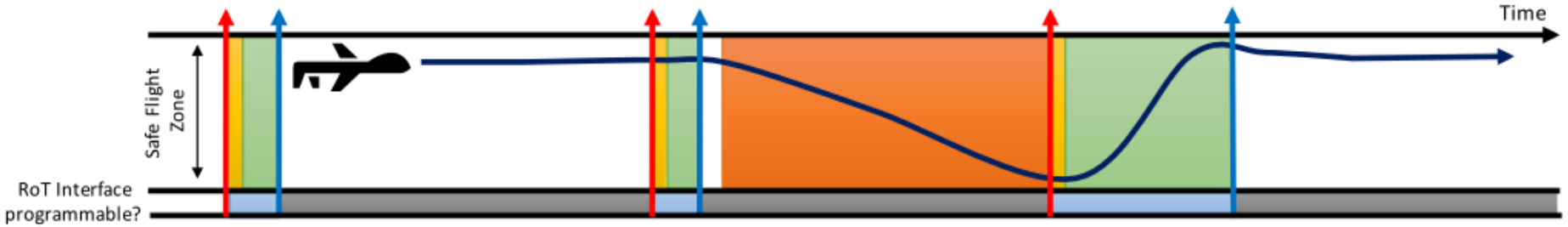


Fig. 1. Example sequence of events for the restart-based implementation of the SEI. White: mission controller is in charge and platform is not compromised. Yellow: system is undergoing a restart. Green: SEI is active, SC and `find_safety_window` are running in parallel. Orange: adversary is in charge. Blue: RoT accepts new restart time. Gray: RoT does not accept new restart time. Red arrow: RoT triggers a restart. Blue arrow: SEI ends, the next restart time is scheduled in RoT, and the mission controller starts.

TEE-based Implementation

- **Removes the restart overhead of the other model**
- **Better for systems with**
  - Higher reliance on memory (RAM wiped on reboot - volatile)
  - Longer reboot times
- **Utilizes TrustZone and LTZVisor to isolate a secure and insecure system**
  - Rather than forcing reboots for every SEI, we force the secure system to run every SEI by giving a higher priority task
- **No RoT required, TrustZone effectively isolates the secure and insecure systems**

TEE-based Implementation

- **Rem**
- **Bett**
  - H
  - L
- **Util** **and inse**
  - R
    sy
- **No l** **e secure and insecure systems**

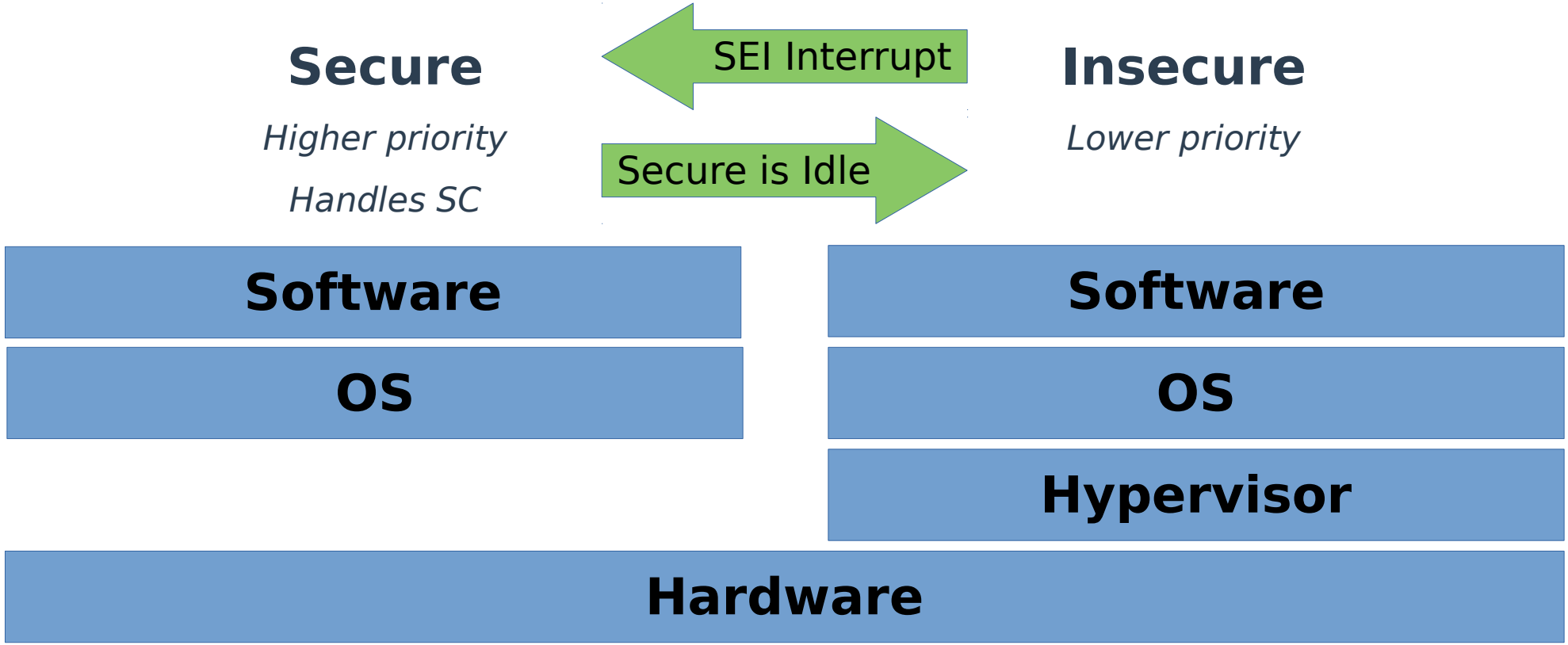
Question by Akinori Kahata:

I cannot understand why TEE has to divide the secure zone and the nonsecure zone. In my understanding, if the architecture can make a secure zone, all of the code can be executed in the secure zone. Why TEE uses the secure zone and nonsecure zone one after the other.

TEE-based Implementation

- **LTZVisor scheduling policy forces the insecure system to run only when the secure system is idle**
- **Secure interrupts bypass TrustZone hypervisor**
  - Meaning the nonsecure system can’t interrupt the secure system immediately without going through the hypervisor and finding an idle moment
  - In contrast, the secure system can interrupt the nonsecure system immediately without a hypervisor middleman
- **Every SEI the secure system is awoken, transitioning the system from insecure to secure**

TEE-based Implementation



TEE-based Implementation

- **Without frequent reboots, compromised systems will remain compromised without one of the following:**
  - Randomized, rare restarts
  - Monitor the platform, during the SEI, for potential intrusions and malicious activities and restart the platform after the malicious behavior is detected

TEE-based Implementation

- **Without frequent reboots, compromised systems will**
  foll
  - R
  - M

ons
a
e
r

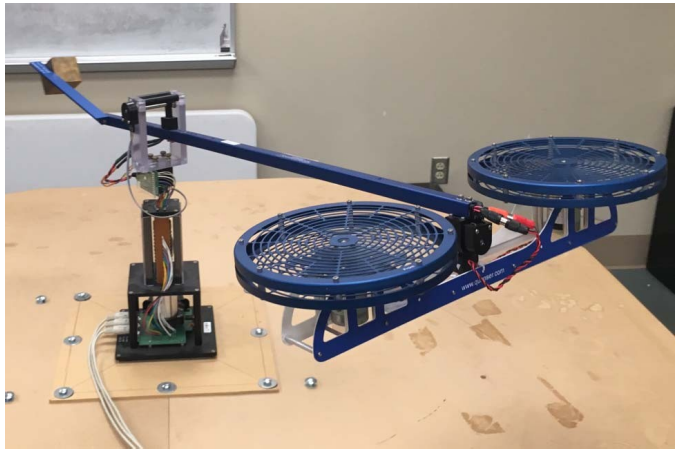
Question by Pat Cody:

Is the performance tradeoff between the restart-based approach vs. spending extra money on TEE-equipped hardware worth it?

Does it work?

Testing

- **Warehouse Temperature Management System**
  - Keep temperature between 20° and 30° C
- **3-Degree of Freedom Helicopter (3DOF)**
  - Prevent fans from hitting the surface below



Warehouse Temperature Management System

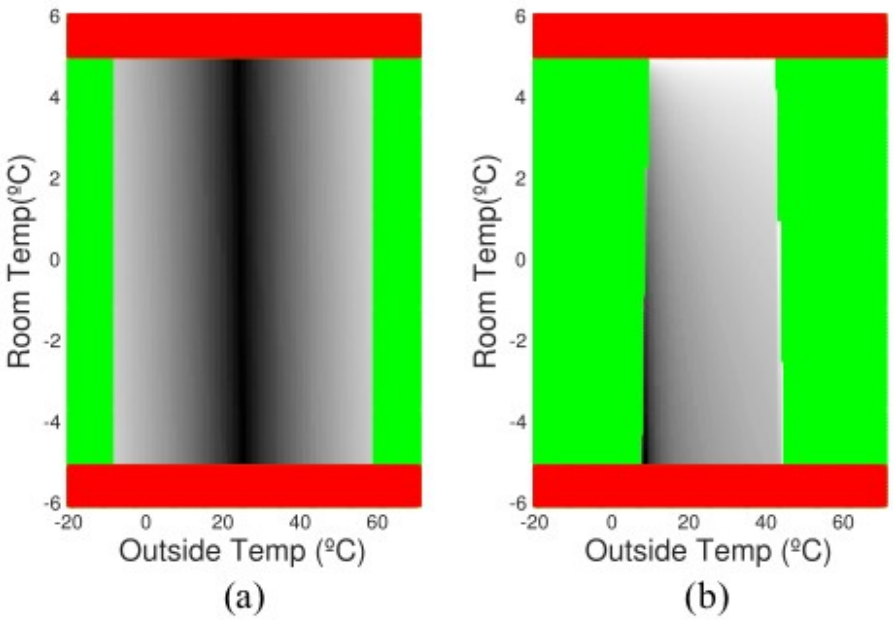


Fig. 3. Safety window values for the warehouse temperature. Largest value of the safety window—the darkest region—is 6235 s. (a) Projection into $T_F = 25°C$. (b) Projection into $T_F = 29°C$.

Red – Inadmissable states/failure

Green – Approaching failure, SC needed but safety not guaranteed

Black/Grey/White – Safe operation, darkness represents how far from instability (darker is safer)

3-Degree of Freedom Helicopter (3DOF)

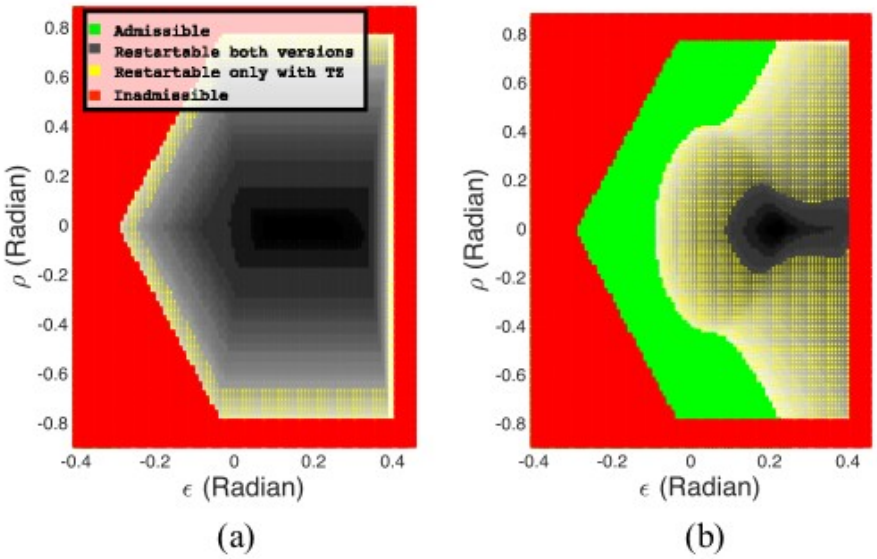


Fig. 4. Safety window values for the 3DOF helicopter. Largest value of the safety window—the darkest point—is 1.23 s. (a) Projection of the state space into the plane $\dot{\epsilon} = 0$, $\dot{\rho} = 0$, $\lambda = 0$, and $\dot{\lambda} = 0.3$ Radian/s. (b) Projection of the state space into the plane $\dot{\epsilon} = -0.3$ Radian/s, $\dot{\rho} = 0$, $\lambda = 0$, and $\dot{\lambda} = 0.3$ Radian/s.

Red – Inadmissable states/failure

Green – Approaching failure, SC needed but safety not guaranteed

Yellow – Safe operation with TEE-implementation

Black/Grey/White – Safe operation, darkness represents how far from instability (darker is safer)

What’s the catch?

Safety Controller vs Traditional Controllers

- **The safety controller is slower and constantly adjusting states to maintain safety**

v_1

v_2

Safety Controller

- Won’t give “complete control” to prevent failure (flipping the drone)
- Tries to remain stable

Traditional Controller

- Gives full control
- Susceptible to failure (flipping)

Paper’s Assumptions

- **Attacker’s Access**
  - Attackers require external interface (such as network, serial port, or debugging interface)
  - No physical access to the platform
  - Once breached, attacker has full control (root access) over the software, actuators, and peripherals

Paper’s Assumptions

- **Platform and Adversary Capabilities**
  - Integrity of Original Software Image
  - Read-Only Storage for the Original Software Image
  - Trusted Execution Environment (TEE)
  - External interfaces (such as debugging interface or network connections) to a device remain disabled on reboot
  - Integrity of Root of Trust
    - The hardware timer that calculated the SEI for the restart-based approach is in fact isolated and secure from the machine

Paper's Assumptions

- **Platform and Adversary Capabilities**
  - I
  - P
  - T
  - E
    C
  - I

Question by Henry Jaensch:

How would updates work on a system like this, if the trusted code is in read only memory?

vork

ed

Paper’s Assumptions

- **Attacks Not Considered**
  - External sensor spoofing or jamming attacks (e.g., broadcasting incorrect GPS signals, electromagnetic interference on sensors, etc.).
  - Data leak related attacks such as those which aim to steal secrets, monitor the activities, or violate the privacy.
  - Network attacks, such as man-in-the-middle or denial-of-service attacks that restrict the network access.

Personal Opinion