# Mixed-Trust Real-Time Computation

Dionisio de Niz

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.



DM23-0262

# Motivation

- Cyber-Physical Systems (CPS) are used in many safety-critical applications
- Verifying these complex systems is a challenging problem
- Current verification techniques do not scale to complex components

**Problem:** How to verify every component and their interaction?

# Motivation

- Cyber-Physical Systems (CPS) are used in many safety-critical applications
- Verifying these complex systems is a challenging problem
- Current verification techniques do not scale to complex components

**Solution:** Do not verify every component but instead use a **mixed-trust framework:**

- Untrusted components interact with trusted (verified) components
- Trusted components guarantee timing and functional correctness of the entire system

# Motivation

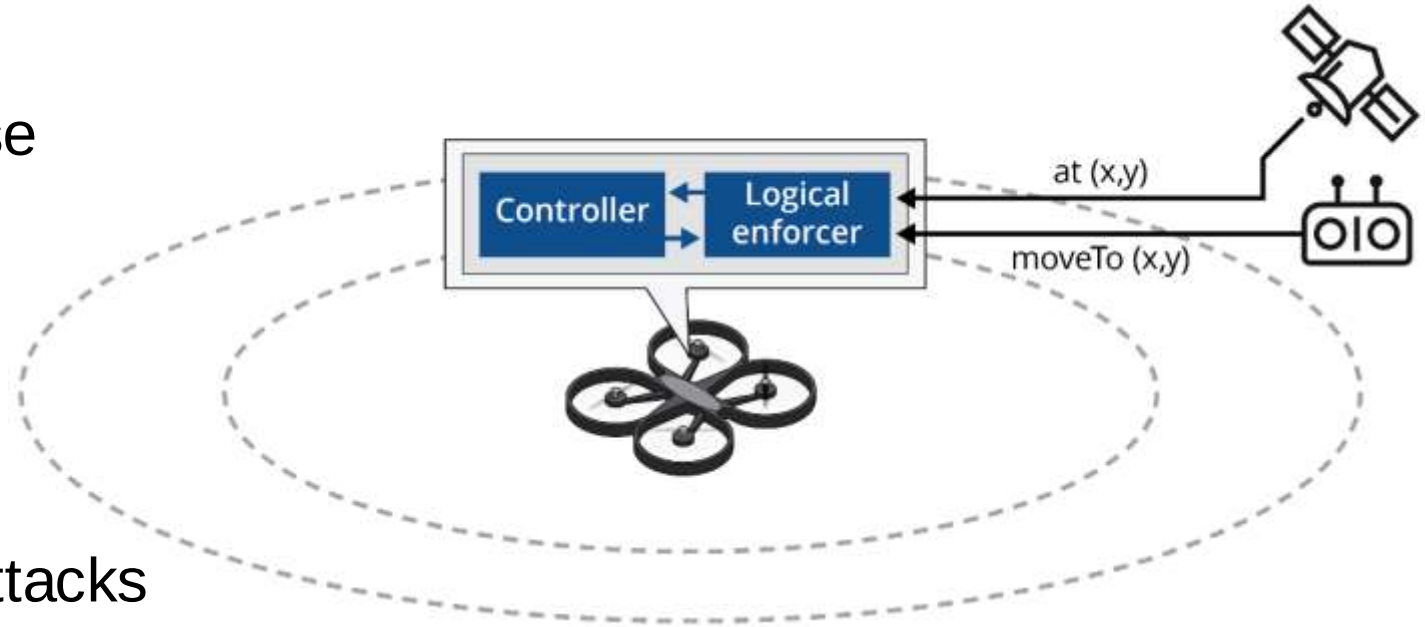
- Cyber-Physical Systems (CPS) are used in many safety-critical applications
- Verifying these complex systems is a challenging problem
- Current verification techniques do not scale to complex components

**Solution:** Do not verify every component but instead use a **mixed-trust framework:**

- **Problem:** Even if trusted components are verified, the system can still be compromised if the interaction between untrusted and trusted components is not verified

# Scalable Enforcement-Based Verification

- Leave Most Code **Unverified**
- Add **simpler (verifiable)** runtime enforcer to make algorithms predictable
- Formally: specify, verify, and compose multiple enforcers
  - Logic: replaces unsafe values
  - Timing: at right time
  - Physics: verified physical effects
- Enforcer protection against failures/attacks
- Resilient to failures / dynamic environment



# Logical Enforcer (Verified)

Statespace & actions

- $S = \{s\}, \phi \subseteq S$
- $R_P(\alpha) \subseteq S \times S;\ R_P(\alpha, s) = \{s' | (s, s') \in R(\alpha)\}$

Enforceable states

- $C_\phi = \{s | \exists \alpha : R_P(\alpha, s) \in C_\phi\}$

Safe actions:

- $SafeAct(s) = \{\alpha | R_P(\alpha, s) \in C_\phi\}$

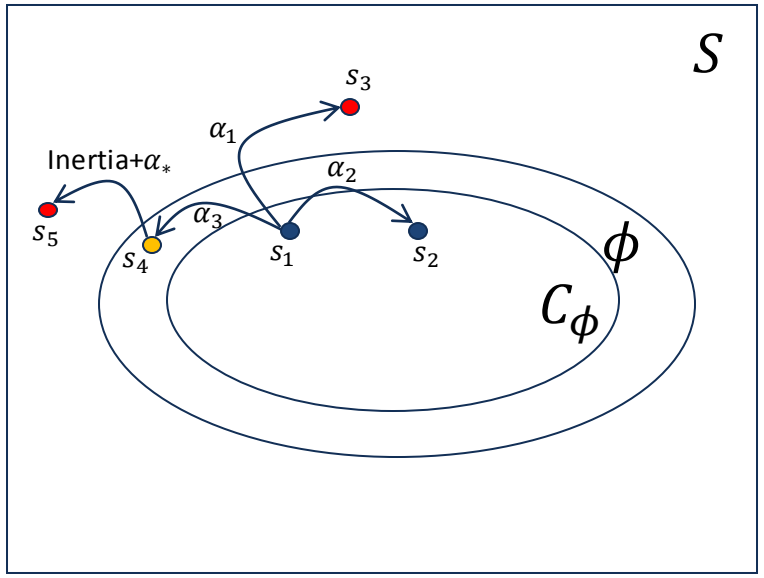
Logical Enforcer: $E = (P, C_\phi, \mu)$

- Set of safe actions:

$$\mu(s) \subseteq SafeAct(s)$$

- Monitor and enforce safe action:

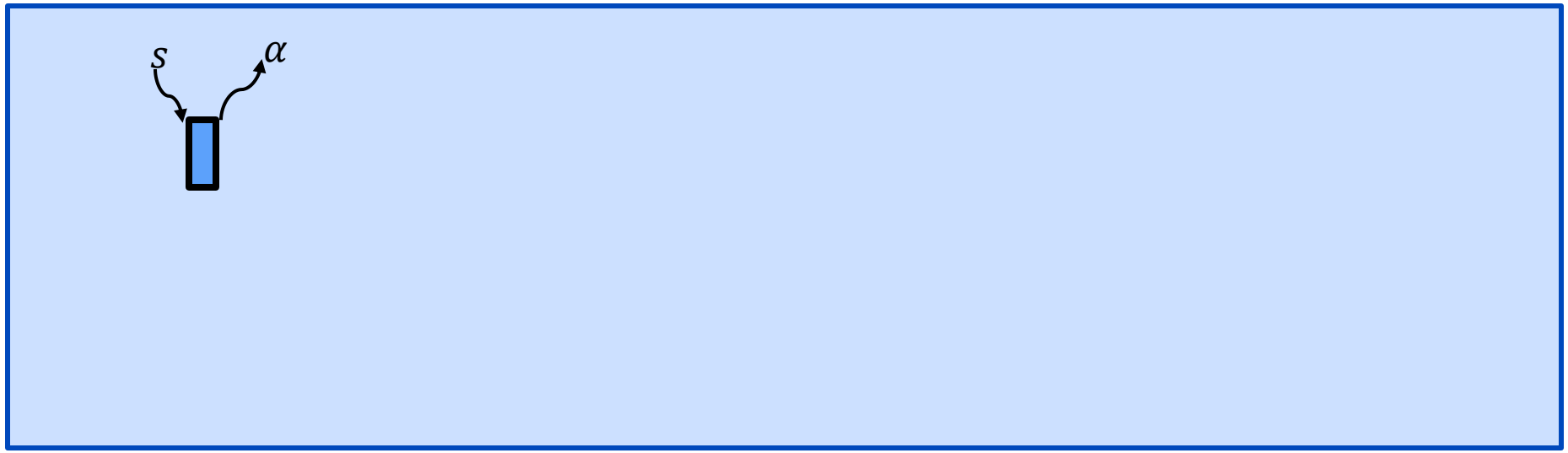
$$\tilde{\alpha} = \begin{cases} \alpha, & \alpha \in \mu(s) \\ pick(\mu(s)), & otherwise \end{cases}$$



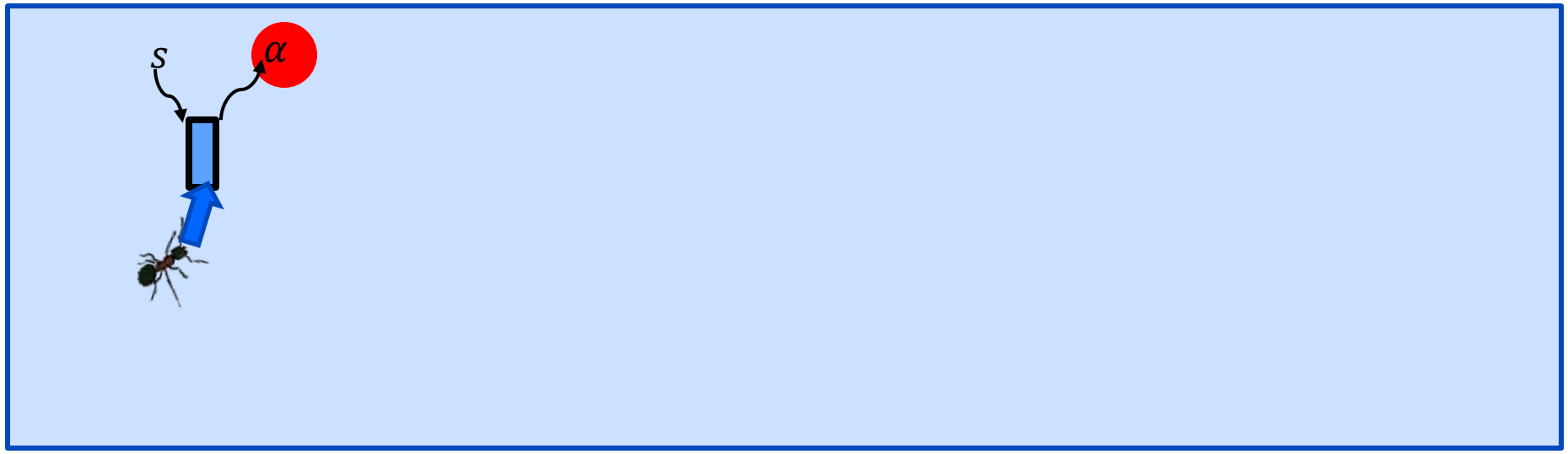

Verification with SMT (Z3)

Bjorn Andersson, Sagar Chaki, and Dionisio de Niz. Combining Symbolic Runtime Enforcers for Cyber-Physical Systems. International Conference in Runtime Verification. 2017

# Enforcing Unverified Components

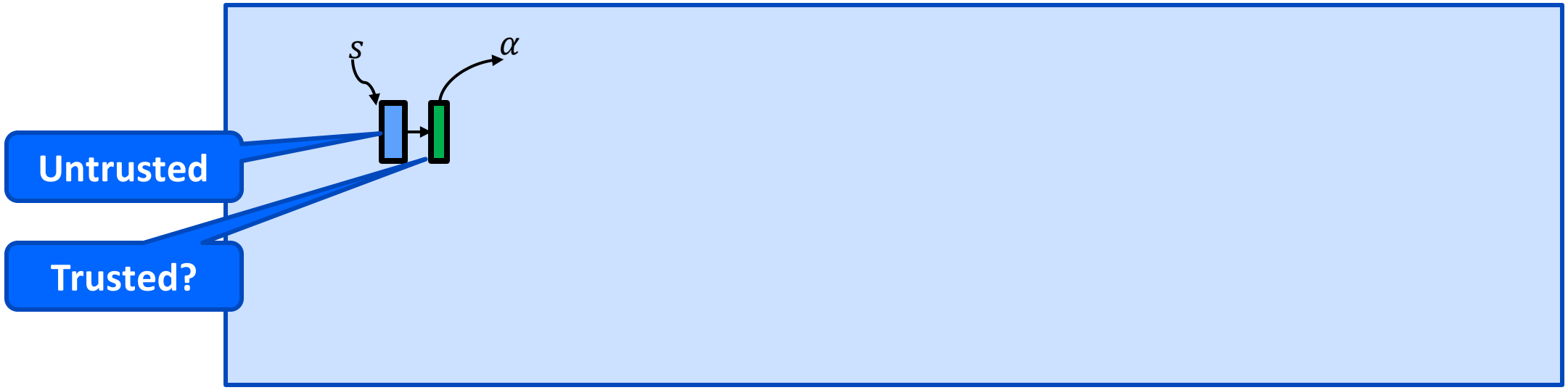


# Enforcing Unverified Components

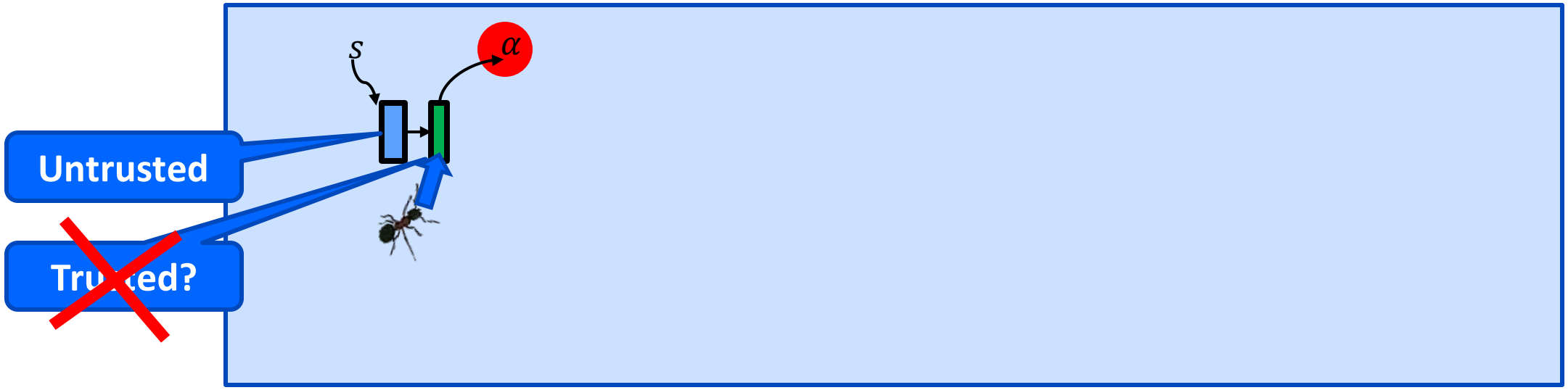



Ant illustration by Jan Gillbank, license by Creative Commons Attribution 3.0 Unported

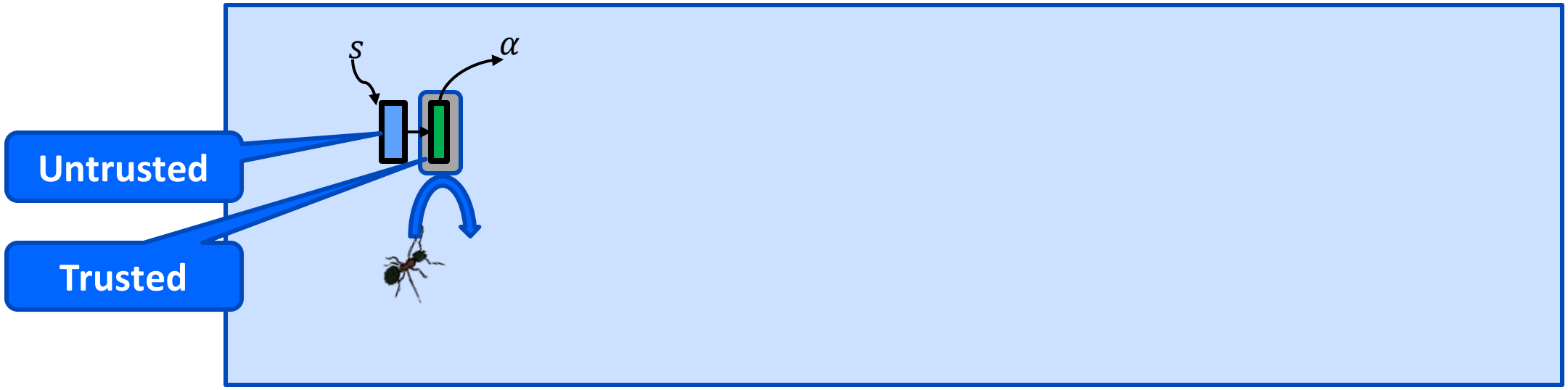
# Enforcing Unverified Components



# But enforcer can be corrupted (bug or cyber attack)

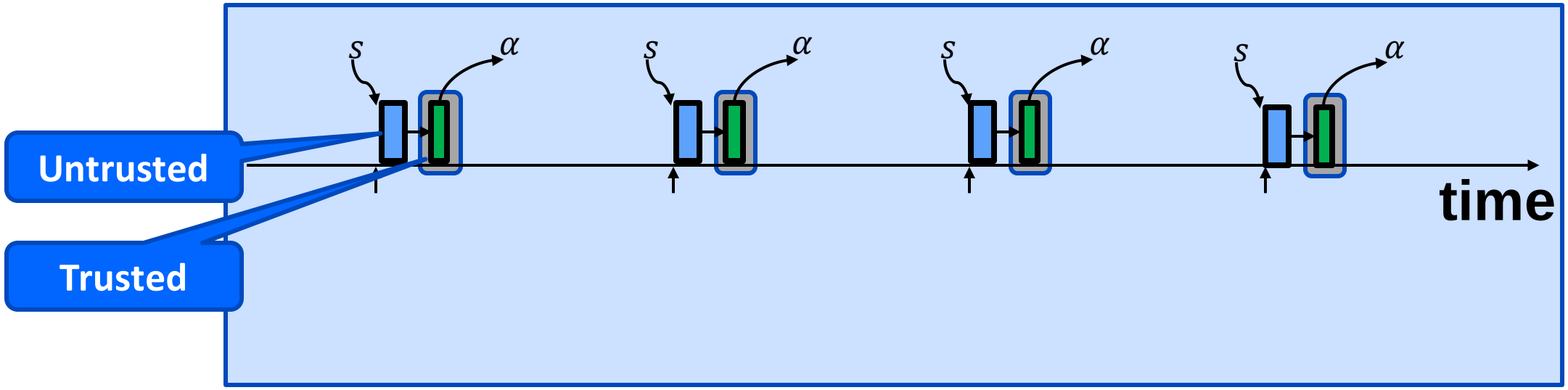


# Add Memory Protection

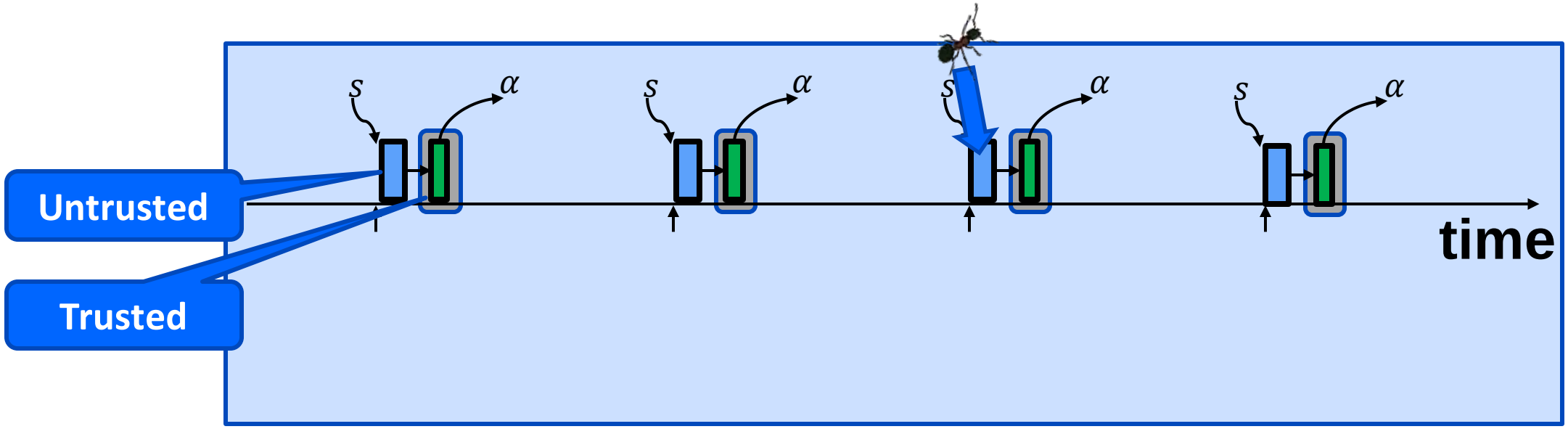



**Trusted = Verified & Protected**

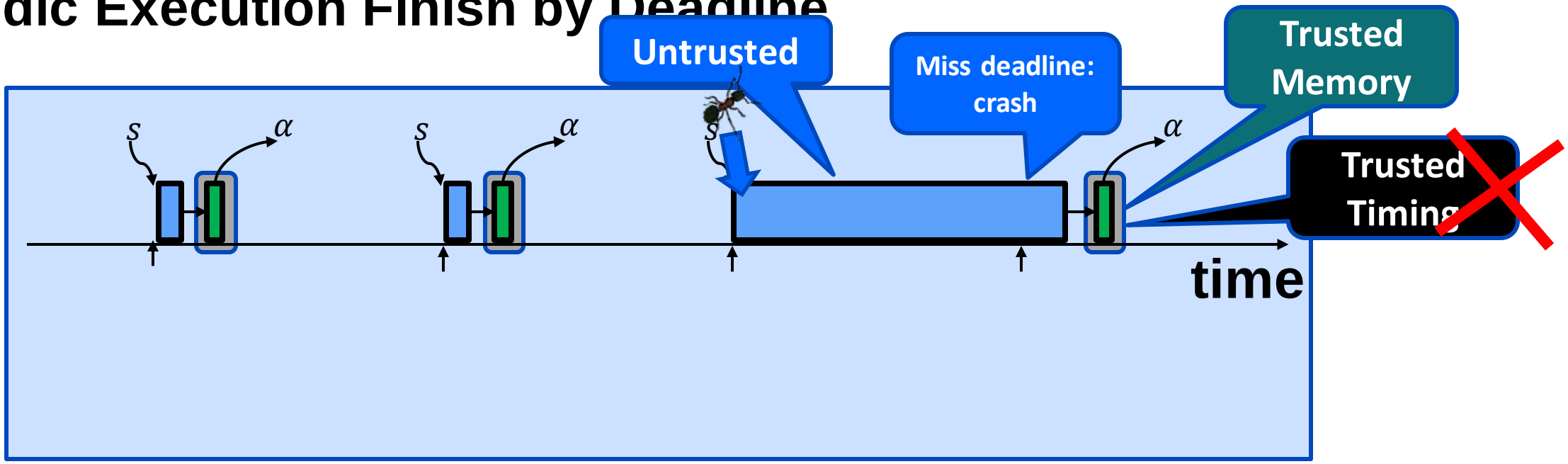
# Periodic Execution Must Finish by Deadline



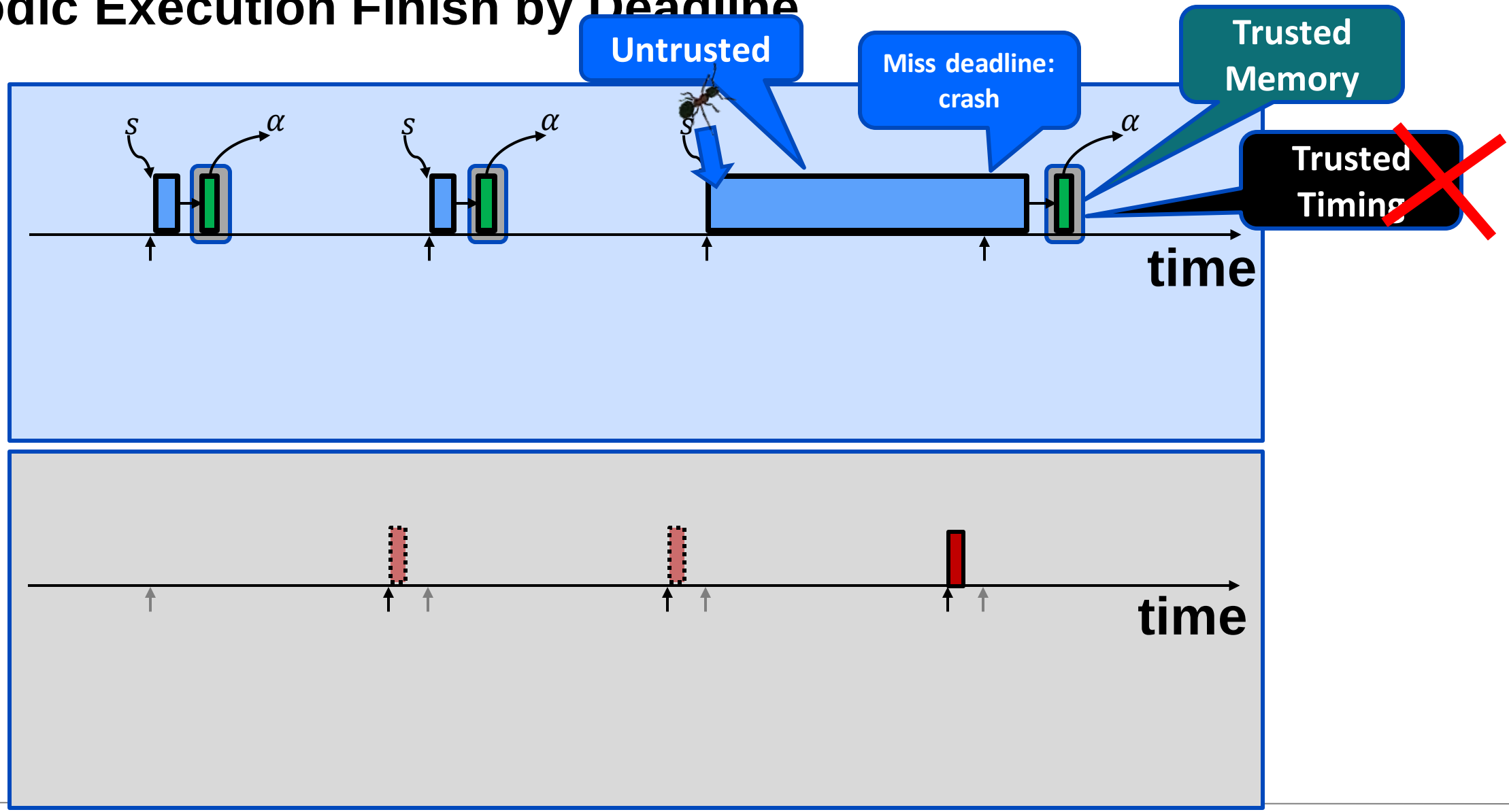
# Periodic Execution Must Finish by Deadline



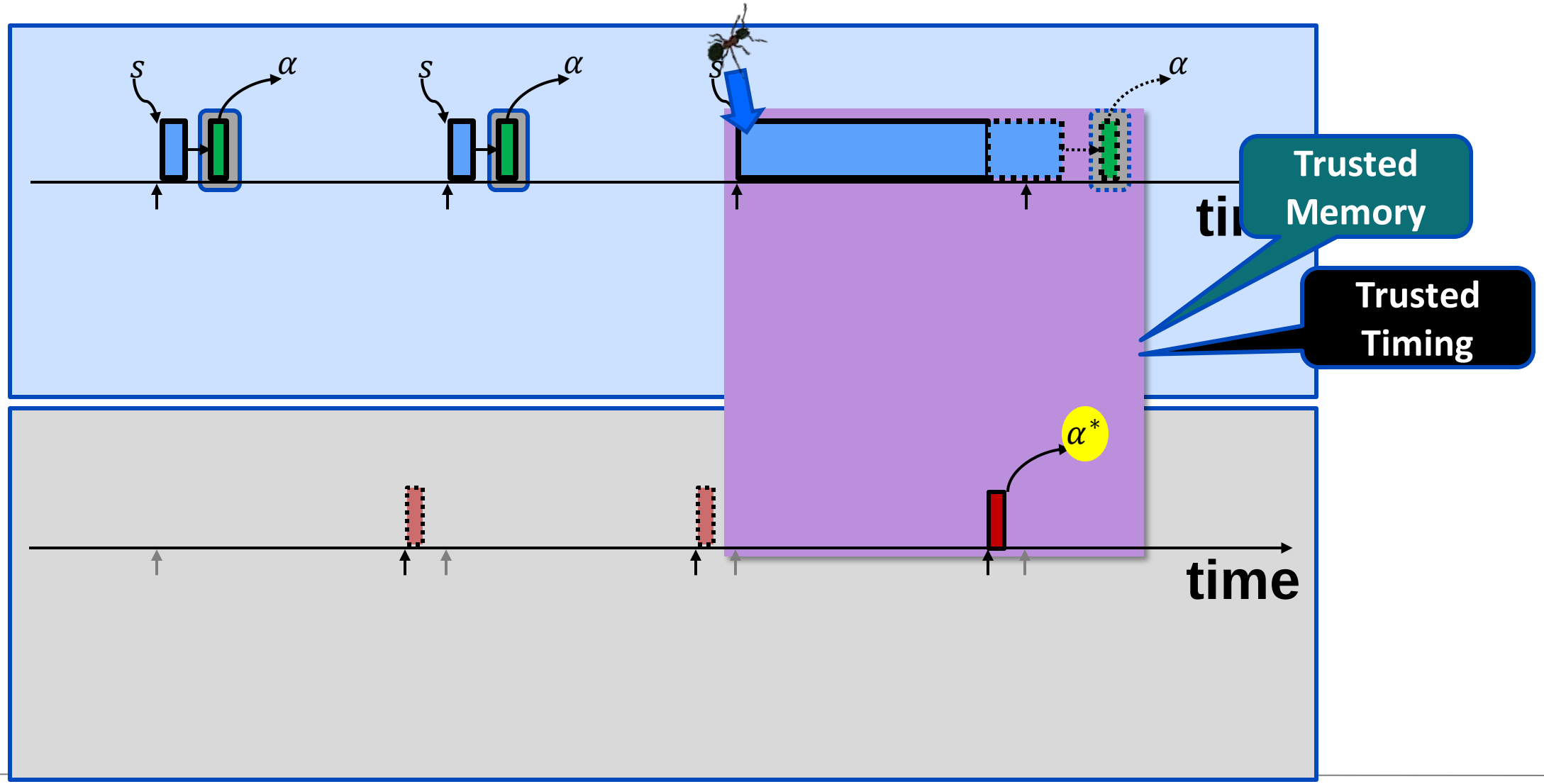
# Periodic Execution Finish by Deadline



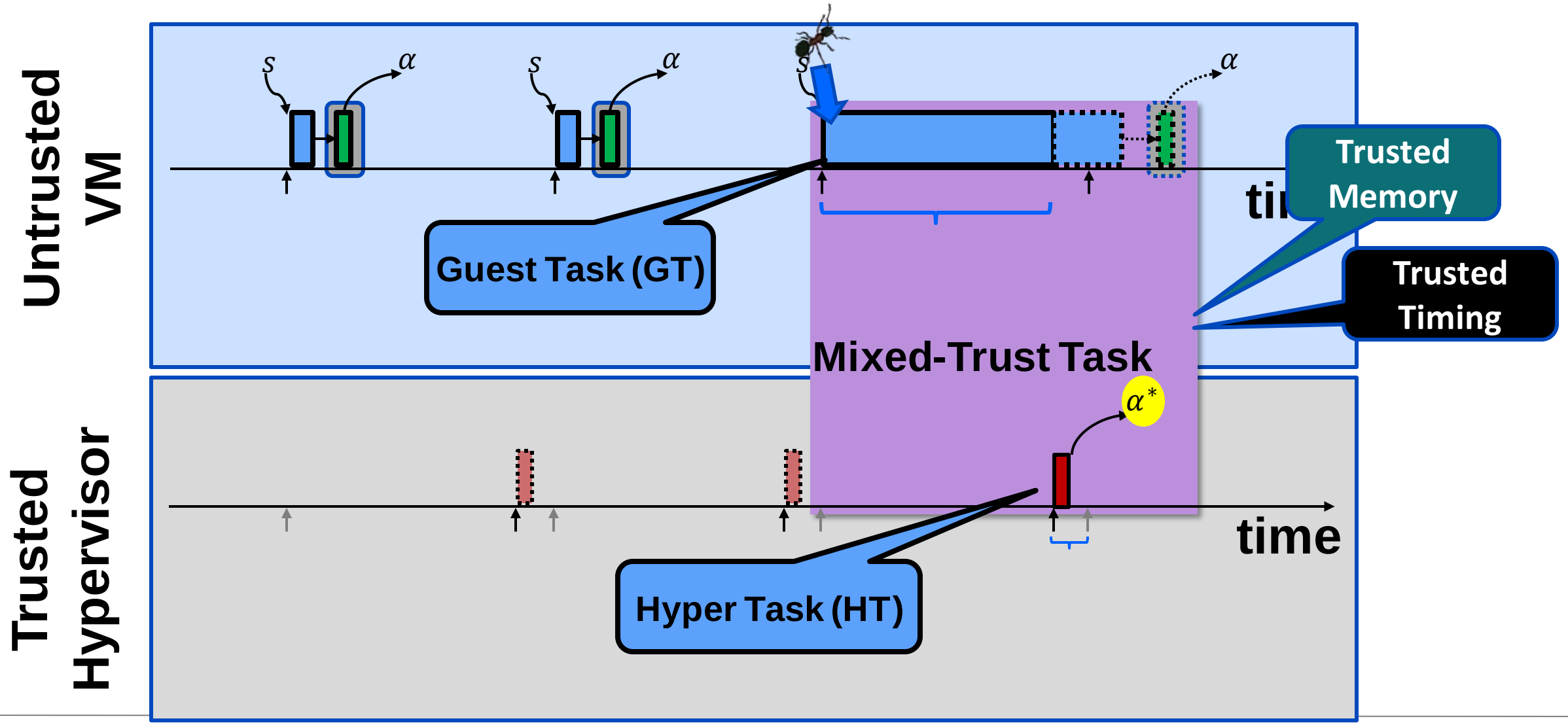
# Periodic Execution Finish by Deadline



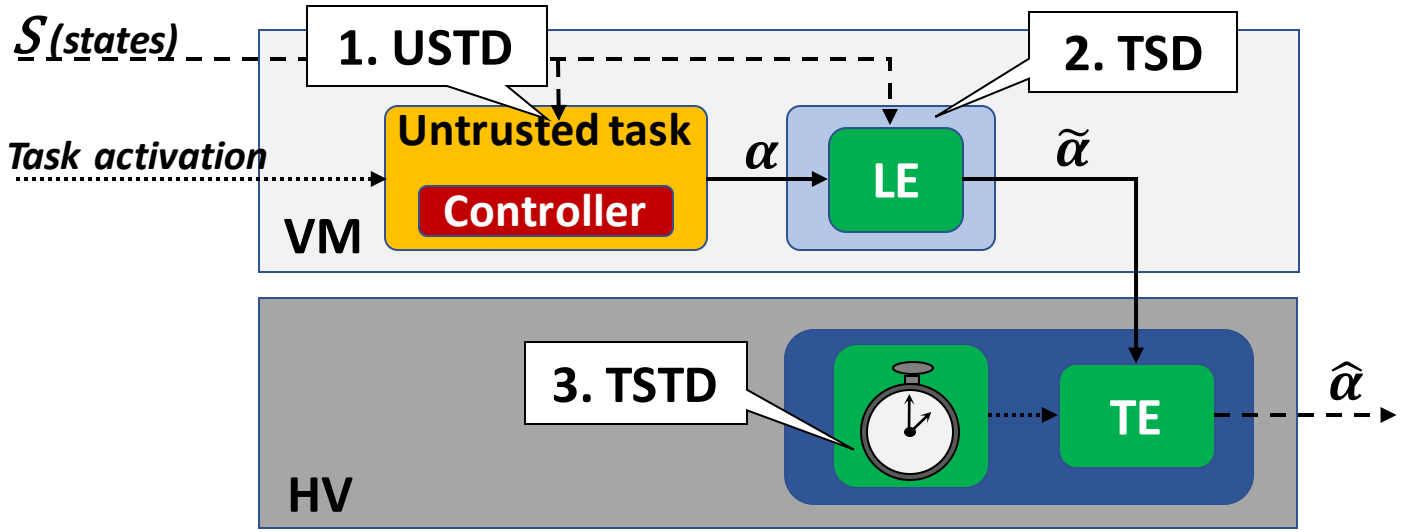
# Periodic Execution Finish by Deadline



# Real-Time Mixed-Trust Computation



# Mixed-Trust Protection Domains




**Untrusted Spatio-Temporal protection Domain (USTD)**

- Untrusted task code execution in the VM

**Trusted Spatial protection Domain (TSD)**

- LE execution in secure enclaves (memory protection)

**Trusted Spatio-Temporal protection Domain (TSTD)**

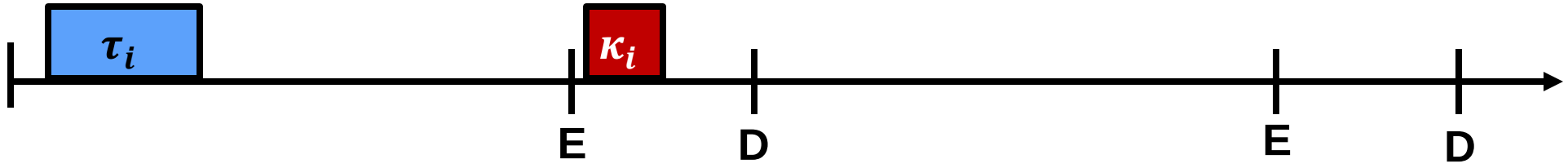
- TE execution in the verified HV (memory and spatial protection)

# Fail-Safe Coordination Protocols

To prevent any dependency of trusted code from untrusted code

1. Secure HT Bootstrapping
   - Required for **Trusted Spatio-Temporal protection Domain (TSTD)**
   - Ensures that HTs can start and execute periodically even if the VM is unable to run GTs
2. Fail-Safe HT Triggering
   - Prevents a failure in the VM from disabling or corrupting the periodic arrival of HTs
3. Late-Output Prevention
   - Prevents the output of a GT $\tau_i$ after its deadline $E_i$

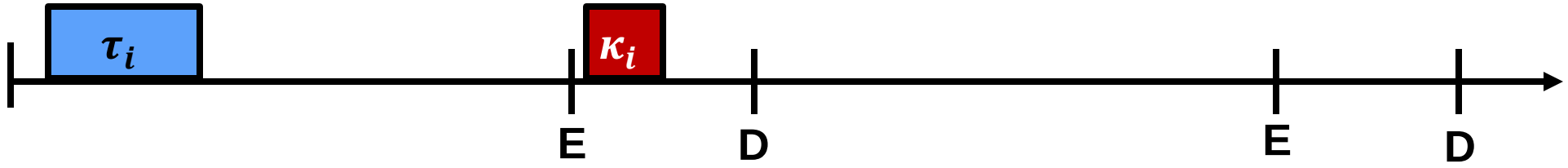
# Mixed-Trust Task Model




Mixed Trust Task

- $\mu_i = (T_i, D_i, \tau_i, \kappa_i)$
  - $T_i$ = Period
  - $D_i$ = Deadline
- Guest Task
  - $\tau_i = (T_I, E_i, C_i)$
    - $E_i$ = Enforcement instant
    - $C_i$ = guesttask execution time
- Hyper Task
  - $\kappa_i = (T_i, D_i, \kappa C_i)$
    - $\kappa C_i$ = hypertask execution time

# Schedulability Equations (1)



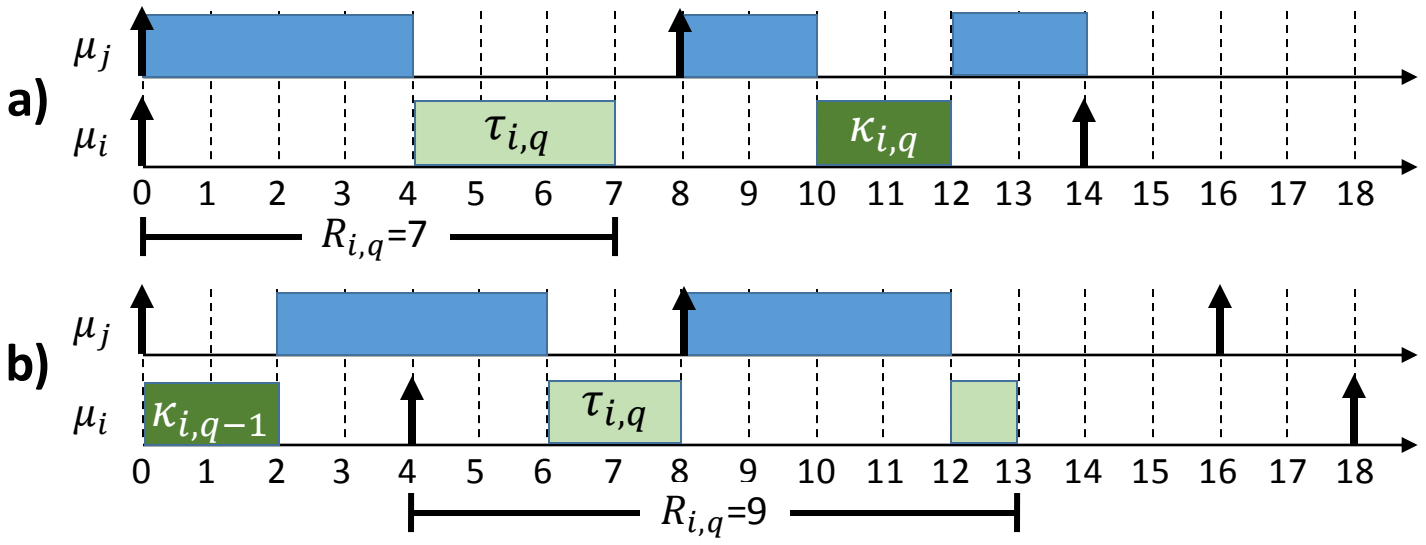

Hypertask : Non-Preemptive

- Level-i active period
  - $t_i^k = \max_{j \in \kappa L_i} \kappa C_j + \left\lceil \frac{t_i^k}{T_i} \right\rceil \kappa C_i + \sum_{j \in H_i} \left\lceil \frac{t_i^k}{T_j} \right\rceil \kappa C_j$
- Starting time of $q$ job
  - $w_{i,q}^k = \max_{j \in \kappa L_i} \kappa C_j + (q-1)\kappa C_i + \sum_{j \in H_i} \left( \left\lfloor \frac{w_{i,q}^\kappa}{T_j} \right\rfloor + 1 \right) \kappa C_j$
- Response time of hypertask
  - $R_i^\kappa = \max_{q \in \left\{ 1 \ldots \left\lceil \frac{t_i^\kappa}{T_i} \right\rceil \right\}} \left( w_{i,q}^\kappa + \kappa C_i - (q-1) T_i \right)$
- Calculate the Enforcement Instant E
  - $E_i = D_i - R_i^\kappa$

# Schedulability Equations (2)

Guest Task Preemptive

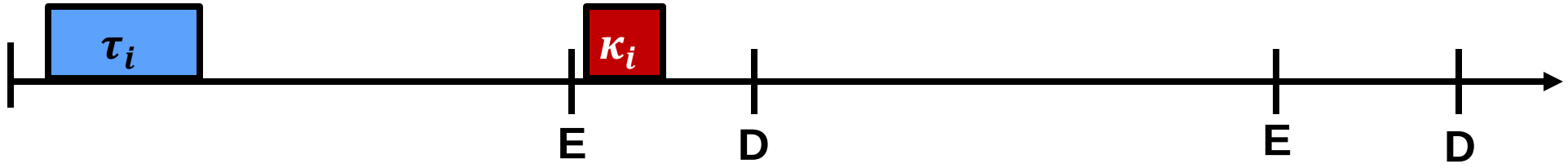
- Need to consider two phasings from HT interference:




**$\tau_i, \tau_j$ arrive at zero**

**$\kappa_i, \tau_j$ arrive at zero**

# Schedulability Equations (3)




Guest Task Preemptive

- Request-bound function (amount of work to perform at this instant)

$$- rbf_i^y(t,b) = \begin{cases} \left\lceil \frac{t-(T_i-E_i)}{T_i} \right\rceil^+ C_i b + \left\lceil \frac{t}{T_i} \right\rceil \kappa C_i, & if\ y = E \\ \left\lceil \frac{t}{T_i} \right\rceil C_i b + \left\lceil \frac{t-E_i}{T_i} \right\rceil^+ \kappa C_i\ , & if\ y = A \end{cases}$$

**Start when $\kappa_i$ arrives**

**Start when $\tau_i$ arrives**

- Active period (for $x \in \{A, E\}$ )
  - $t_i^{g,x} = (\sum_{j \in L_i} rbf_j^E(t_i^{r,x}, 0)) + rbf_i^x(t_i^{g,x}, 1) + \sum_{j \in H_i} \max_{y \in \{E,A\}} rbf_j^y(t_i^{g,x}, 1)$
- Start of job (for $x \in \{A, E\}$ )
  - $w_{i,q}^{g,x} = \left(\sum_{j \in L_i} rbf_j^E\left(W_{i,q}^{g,x}, 0\right)\right) + qC_i + (q - 1 + I_{x=E}\kappa C_i + \sum_{j \in H_i} \max_{y \in \{E,A\}} rbf_j^y(w_{i,q}^{g,x}, 1)$
- Response time: (for $x \in \{A, E\}$ )
  - $R_{i,q}^{g,x} = w_{i,q}^{g,x} - \left((q-1)T_i + I_{x=E}(T_i - E_i)\right)$

# Mixed-Trust Synchronization

To prevent an untrusted component from causing unexpected behavior the following conditions must be enforced:

| Conditions |
| --- |
| **C1:** Each mixed-trust task must produce an output every period |
| **C2:** There is only one output per period |
| **C3:** The output produced by a task in a period is either from the LE or TE |
| **C4:** A guest output must be the product of a computation that executes within a single period |
| **C5:** The TE of a task must execute E times units after the arrival of the job it guards and finishes before the end of the period |

**Note:** Only **C5** was formally proven before!

# Mixed-Trust Synchronization

In this talk:

- We address the gap in prior work where conditions C1-C4 were not formally proven
- Additionally, we also prove a new condition C6:

**C6:** If the guest task finishes before the deadline, then the output of this task is used as the output of the mixed-trust task

- C6 with C4 (guest output is the product of a computation in a single period) guarantees that, if there is a **valid run of a guest task**, then that **output is used** instead of the hyper task.

# Timing Semantics

- **Guest job:** a job run by the guest task
- **Hyper job:** a job run by the hyper task

- **Valid guest job:**
  - The job started after the beginning of the current period and finished before the deadline for the guest task
  - If more than one guest job was run in the same period, we only consider as valid the first job

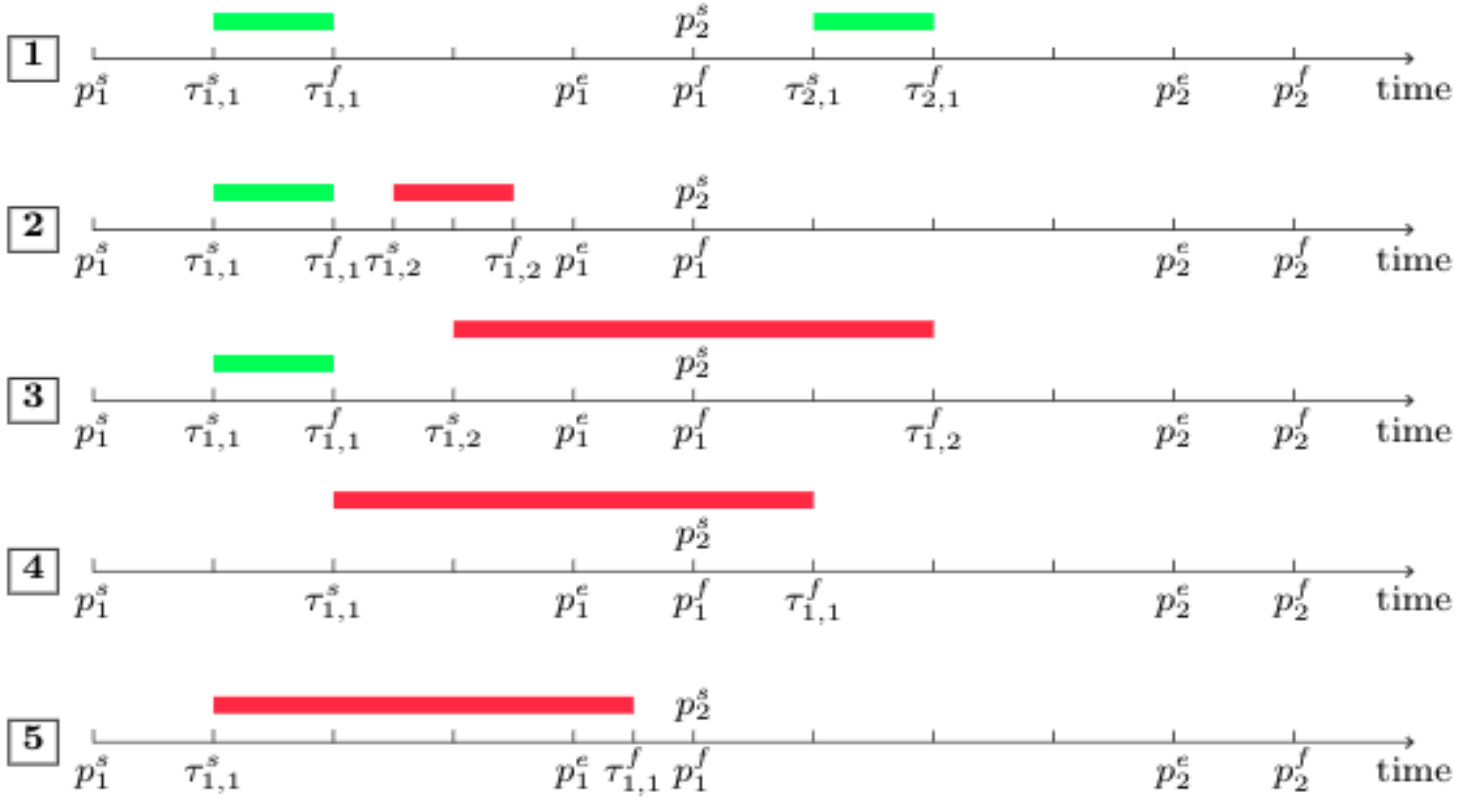
# Timing Semantics

- **Guest job:** a job run by the guest task
- **Hyper job:** a job run by the hyper task

- **Valid execution:**
  - The hyper task only runs one hyper job in the same period and always terminates before the end of the period
  - If there are **no valid** guest jobs:
    - Output of the period is the **output** of the **hyper task**
  - If there is a **valid** guest job:
    - Output of the period is the **output** of the **guest task**

# Timing Semantics

- Example of valid and invalid guest jobs:



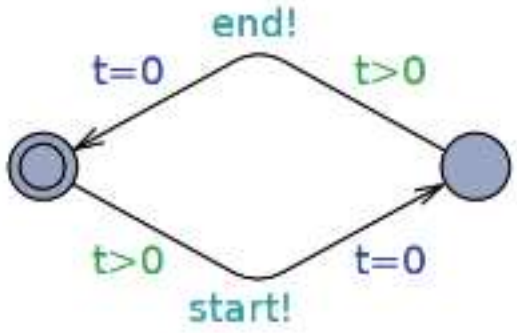
# Modeling the Synchronization Protocol

- We will use UPPAAL to model this synchronization protocol and prove the correctness properties

- Timed automaton:
  - Finite-state machine extended with clock variables where all clock variables progress synchronously

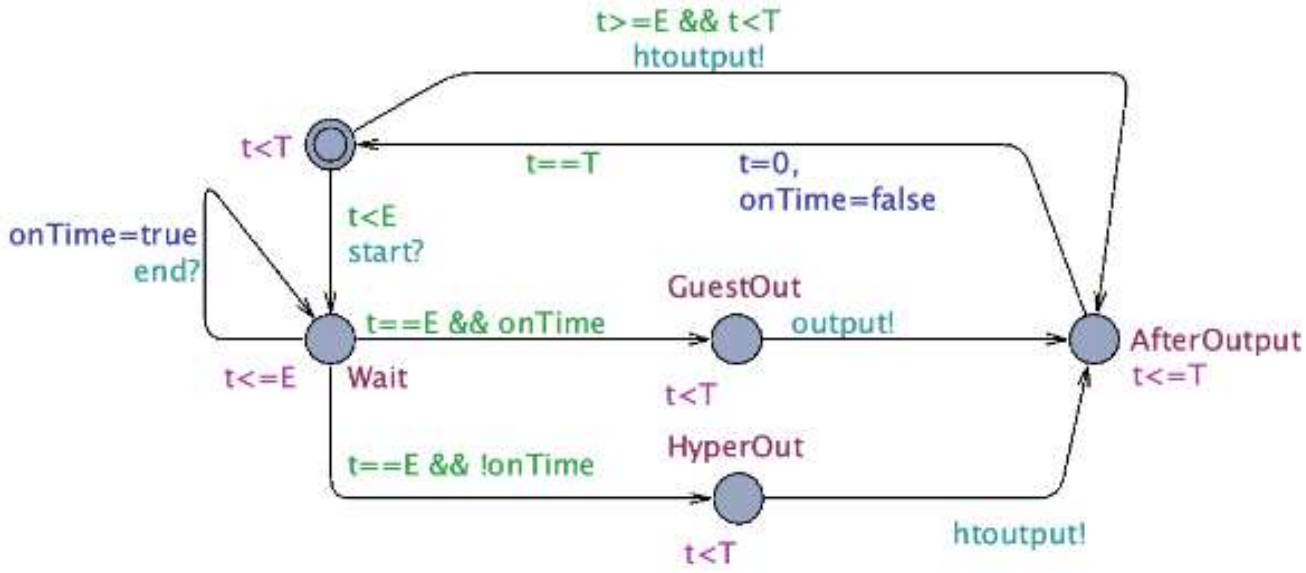
- UPPAAL:
  - Modeling language is an extension of timed automata
  - Bounded discrete variables for additional state information
  - Broadcast synchronization channels

# Guest Task Automaton



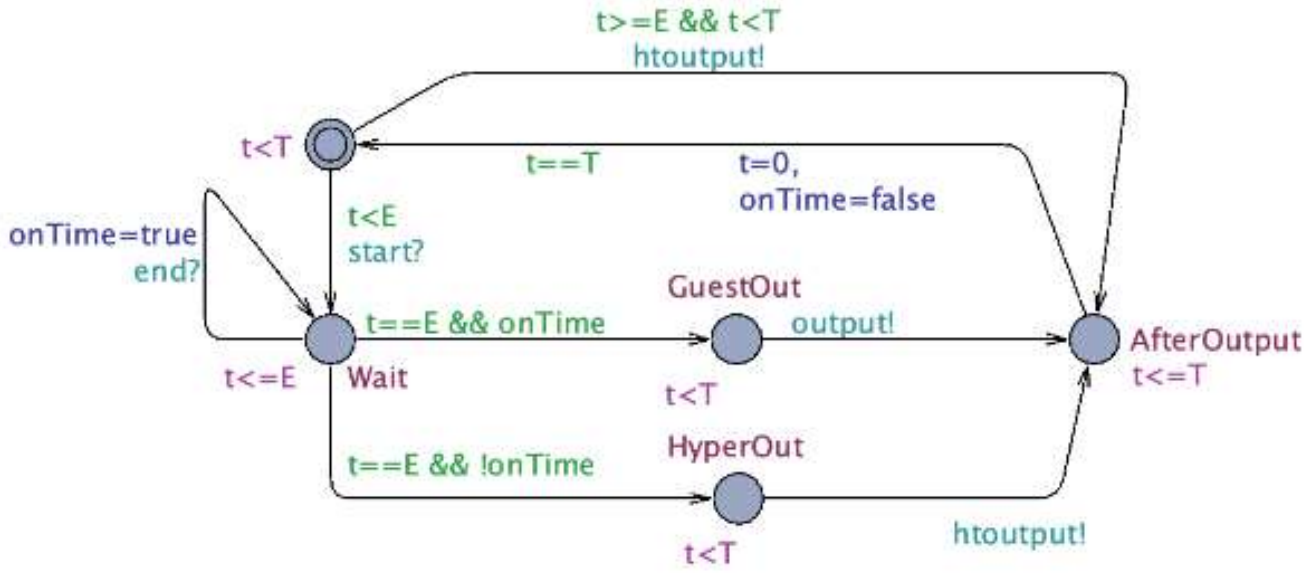

- We do not trust the guest task
- We model its behavior with an automaton that represents all possible behaviors
- Edges:
  - Pre-condition (green): e.g., t>0
  - Post-condition (blue): e.g., t=0
- Broadcast channels:
  - start!
  - end!

# Hyper Task Automaton



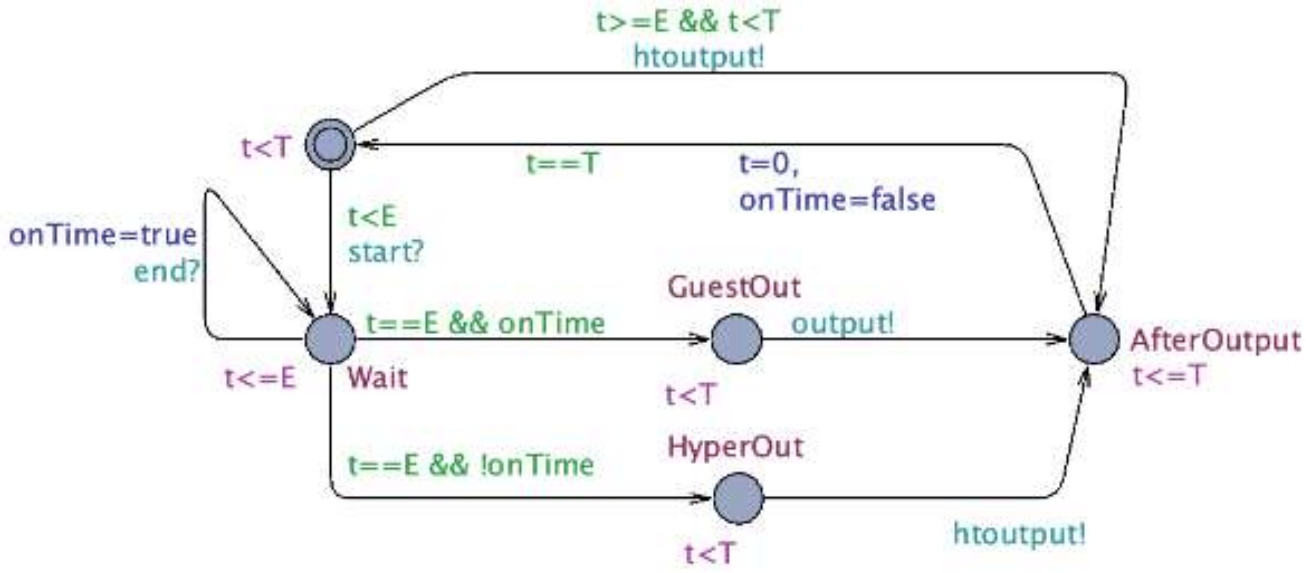

- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**
- Case 1:
  - The start signal is never received

# Hyper Task Automaton



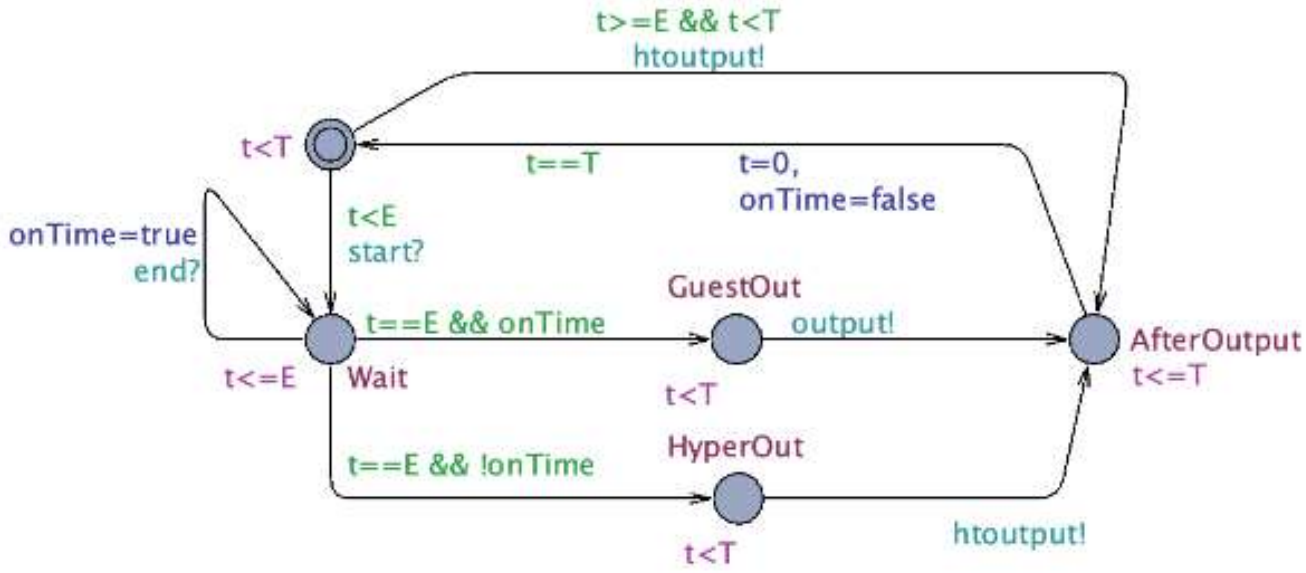

- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**
- Case 2:
  - The start signal is received before the clock reaches E
  - Hyper task transition to the wait location where it waits for an end signal

# Hyper Task Automaton




- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**
- Case 2a:
  - End signal is received before the clock reaches E
  - **valid guest task execution**

# Hyper Task Automaton



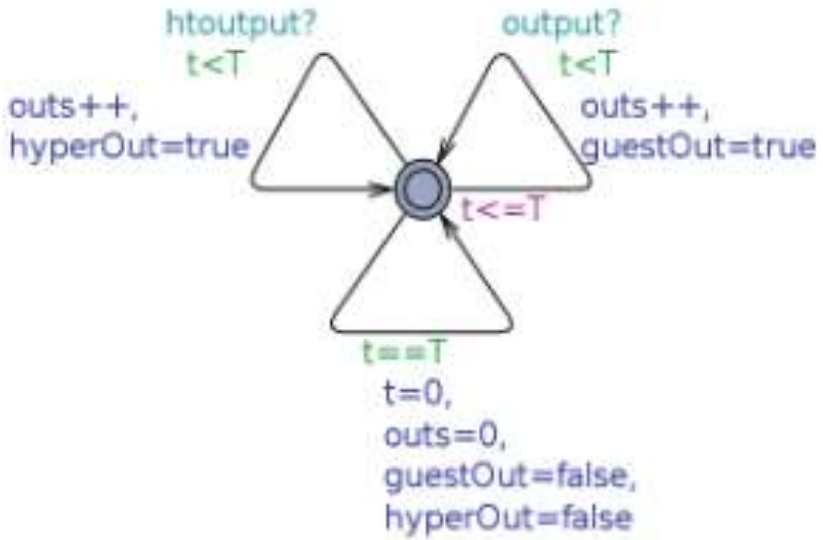

- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**
- Case 2b:
  - End signal is not received before the clock reaches E
  - **invalid guest task execution**

# Verification of Timing Properties

- Properties to be proven:
  - **P1.** Each mixed-trust task must produce an output every period
  - **P2.** There is only one output per period
  - **P3.** If the guest job execution is valid, the output is from the GT
  - **P4.** If the guest job execution is invalid, the output is from the HT

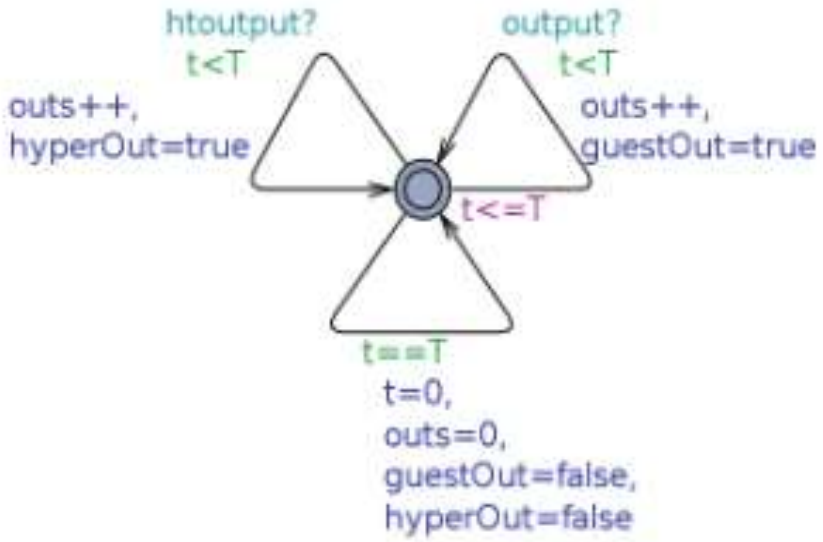
- UPPAAL's specification language is restricted:
  - No alternating temporal operators
  - **Solution:** construct two observer automata that observe the behavior of the guest and hyper task:
    - Easier to write temporal specifications using observers

# Output Observer Automaton




- Behaviors related to P1 and P2 (exactly one output per period) are tracked by the output observer automaton
- We want to prove safety properties, i.e. nothing bad will happen
- Temporal formulas will take the form AG ¬F:
  - ¬F holds in the **current state** and in **all paths of execution**

# Output Observer Automaton



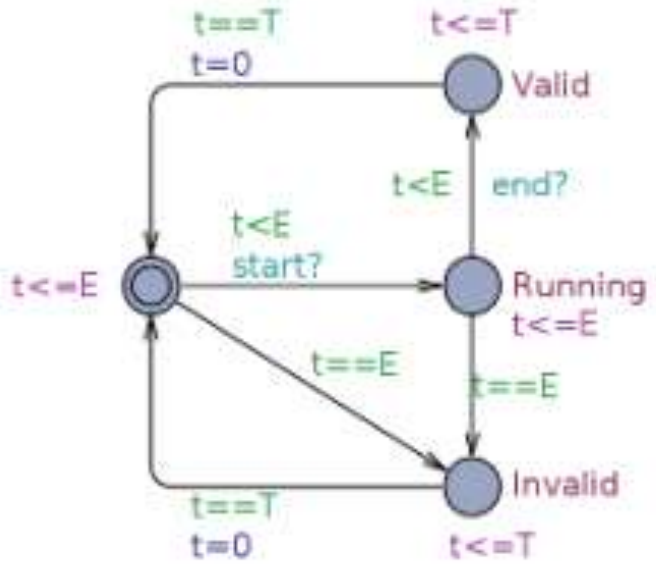

- Behaviors related to P1 and P2 (exactly one output per period) are tracked by the output observer automaton

$$AG \neg (outputObserver.t == T \wedge outputObserver.outs == 0)$$

$$AG \neg (outputObserver.outs > 1 \wedge outputObserver.t < T)$$

- If these formulas are valid then it is guaranteed that the protocol never finishes a period with zero outputs or with more than one output

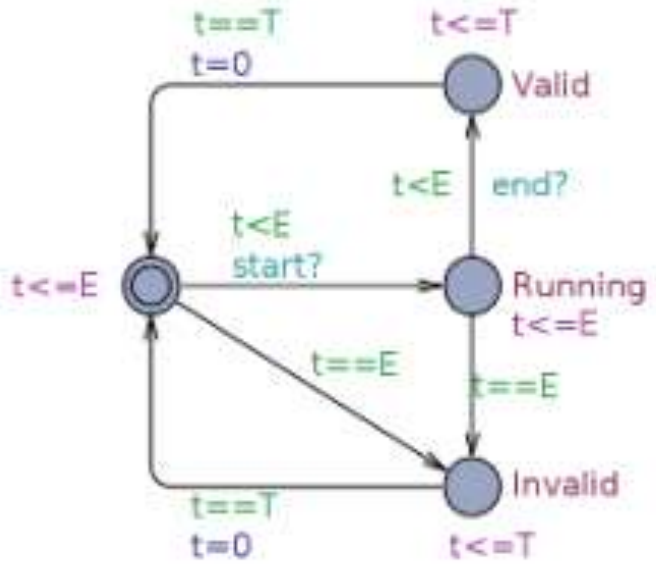
# Job Observer Automaton




- Behaviors related to P3 and P4 (output of the mixed-trust task) are tracked by the job observer automaton

- The signals *start* and *end* from the guest task are tracked to determine if the guest task job execution is valid:
  - **Valid**: if the *start* and *end* signals are received (in that order) within one period before the clock reaches *E*
  - **Invalid**: otherwise

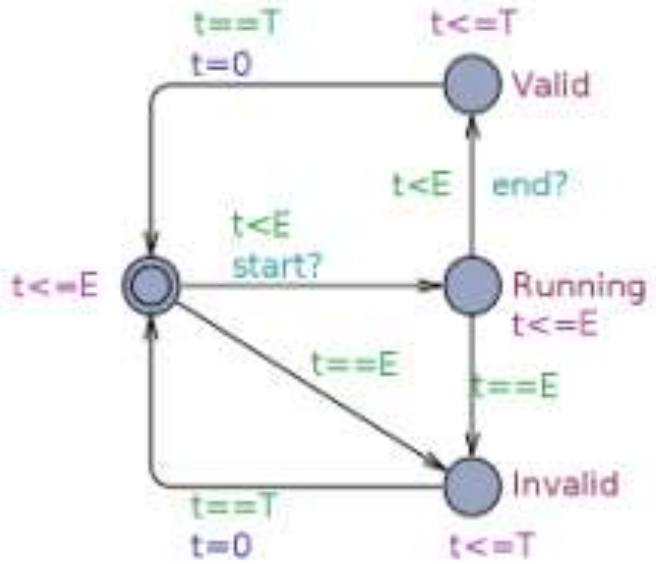
# Job Observer Automaton




- Behaviors related to P3 and P4 (output of the mixed-trust task) are tracked by the job observer automaton
- If the guest job is **valid** then the **output** must be the one from the **guest task**:

$$AG \neg (jobObserver.Valid \wedge hyperOut)$$

# Job Observer Automaton




- Behaviors related to P3 and P4 (output of the mixed-trust task) are tracked by the job observer automaton
- If the guest job is **invalid** then the **output** must be the one from the **hyper task**:

$$AG \neg (jobObserver.Invalid \wedge guestOut)$$

# Proving Temporal Properties

- Properties to be proven:
  - **P1.** Each mixed-trust task must produce an output every period
  - **P2.** There is only one output per period
  - **P3.** If the guest job execution is valid, the output is from the GT
  - **P4.** If the guest job execution is invalid, the output is from the HT

| Property | TCTL Formula | Time (s) |
|---|---|---|
| P1 | $AG \neg (outputObserver.t == P \wedge outputObserver.outs == 0)$ | 57 |
| P2 | $AG \neg (outputObserver.outs > 1 \wedge outputObserver.t < P)$ | 54 |
| P3 | $AG \neg (jobObserver.Valid \wedge hyperOut)$ | 52 |
| P4 | $AG \neg (jobObserver.Invalid \wedge guestOut)$ | 24 |

# Protocol Implementation Corrections

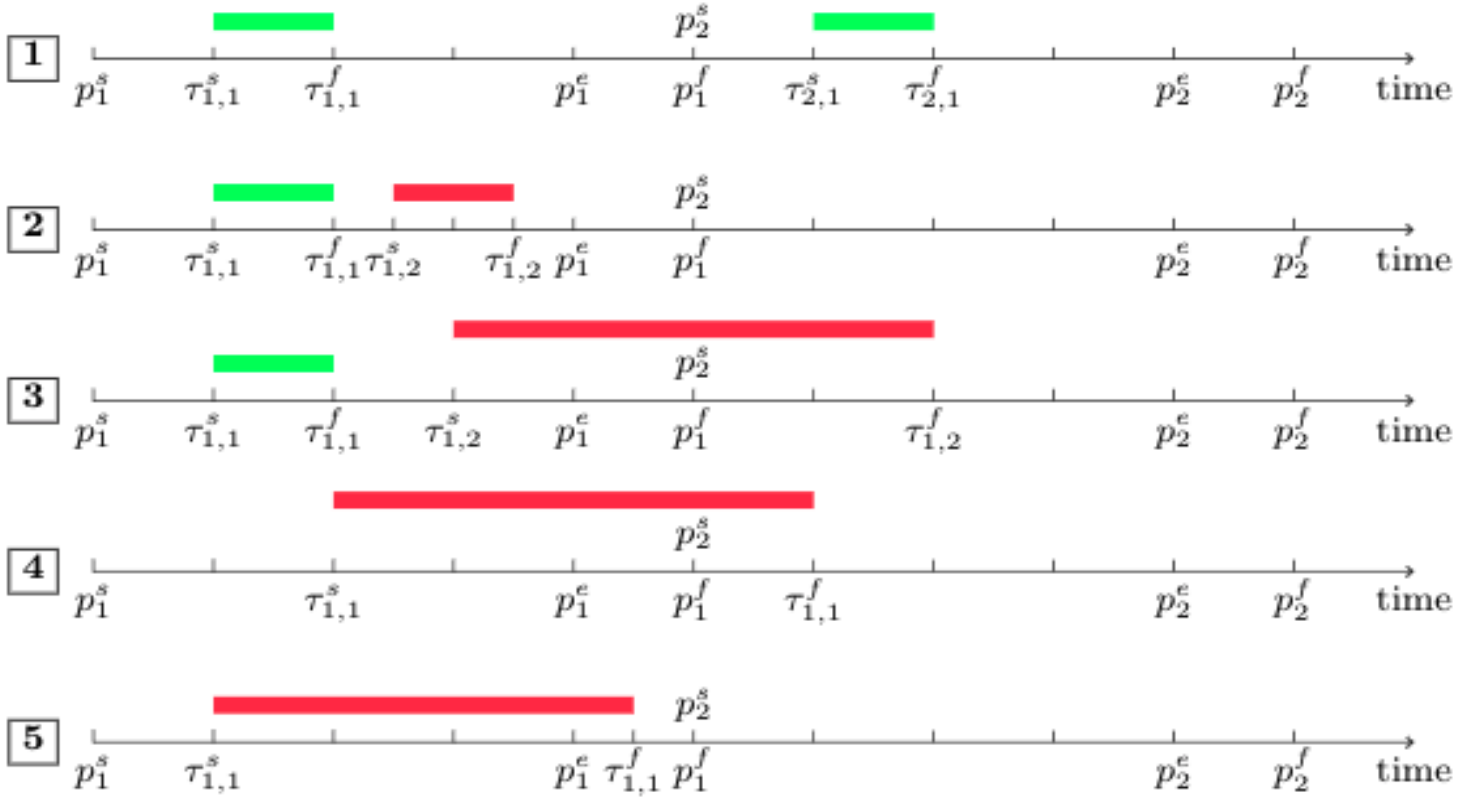
- When modeling this protocol in UPPAAL we found a critical flaw in a previous implementation

- Our modeling and verification properties are ***tightly connected*** with detecting a ***start*** and ***end*** of a guest job

- Previous implementation:
  - Different methodology to detect the completion of a job
  - It was only tracking the ***end*** of the job
  - **Issue:** when a job ends, we do not know if it started in the current period or in the previous period
  - **Breaks:** "**C4:** A guest output must be the product of a computation that executes within a single period"

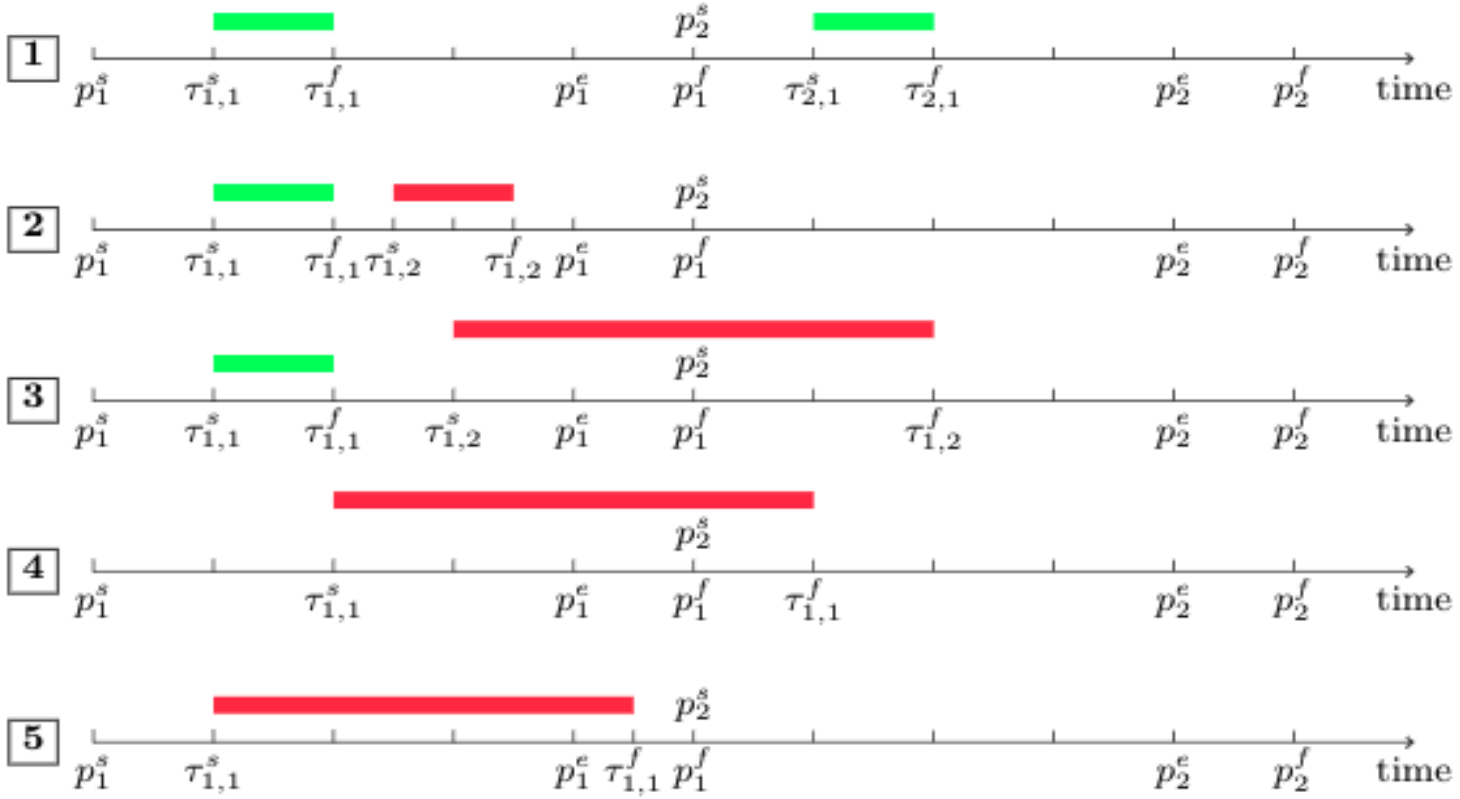
# Protocol Implementation Corrections

- **Bug:** not able to detect the invalid jobs in cases 3 and 4



# Protocol Implementation Corrections

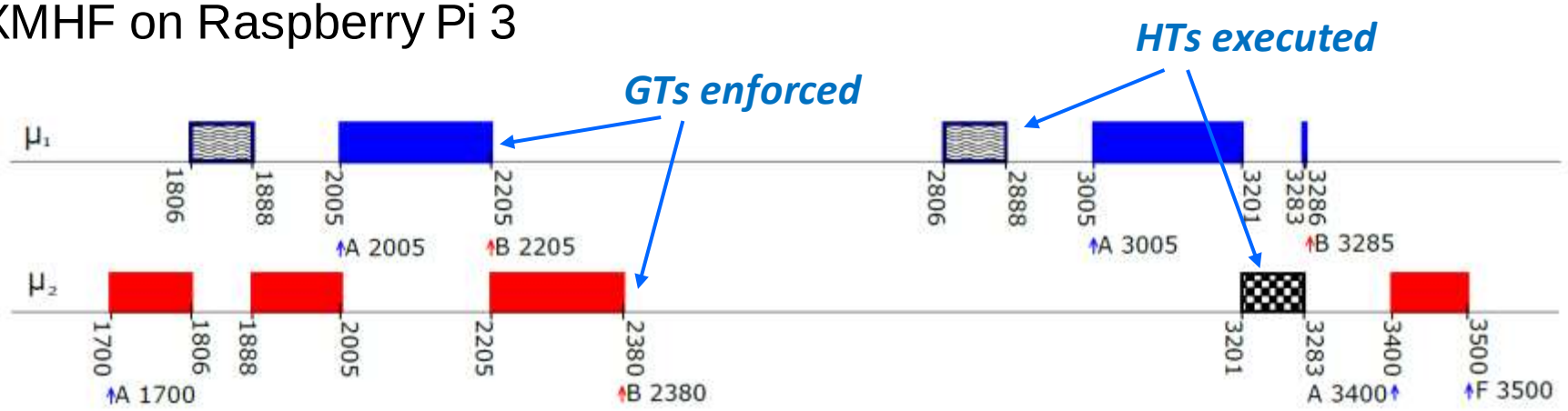
- **Solution:** change the implementation to track **start** and **end** of a job



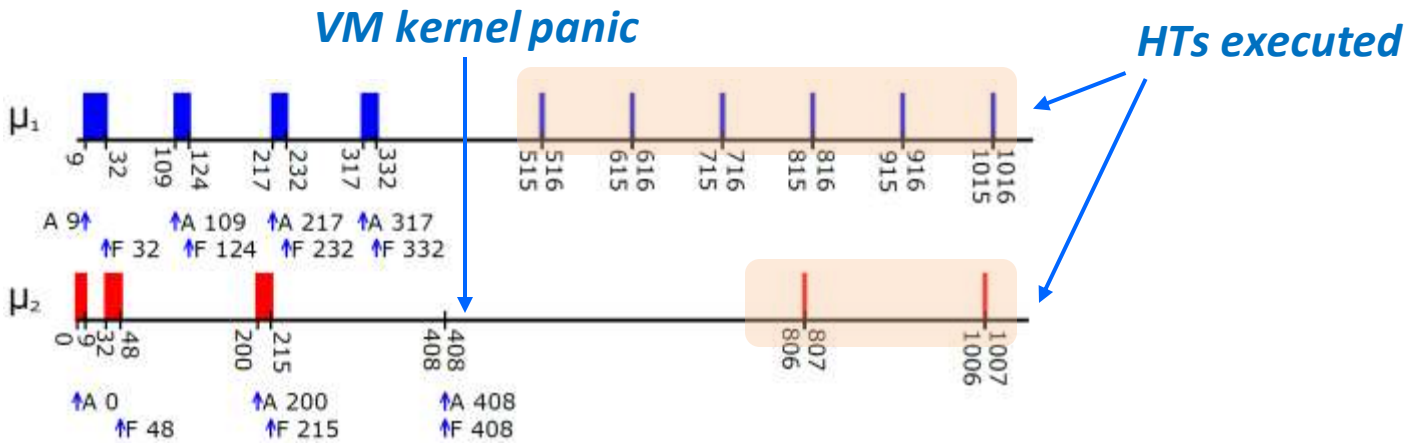
# Case Study: Temporal Failure Scenarios

Implemented in uberXMHF on Raspberry Pi 3

Transient Failures




Permanent Failures

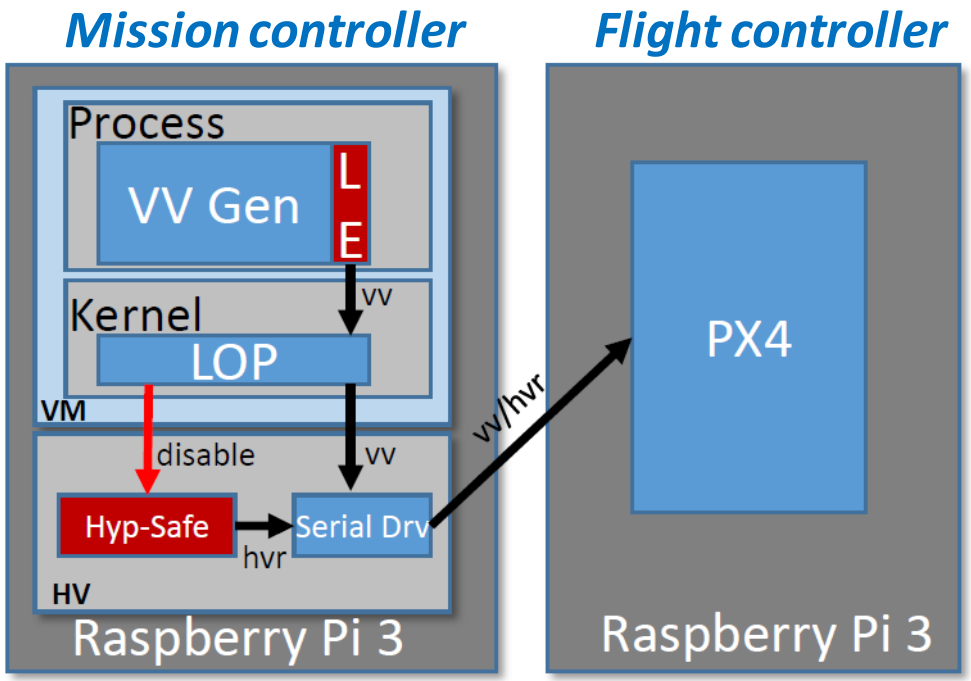


# Case Study: Drone Application

Mission controller: sends velocity vectors (VV) to Flight controller

- Guest task (VV Gen) generates velocity vectors
- Hyper task (Hyp-Safe) generates the safe drone action

Tested with hardware-in-the-loop simulation



# Control-Theoretic Verification

**Recoverable Set:** $\varepsilon_{SCj}(1)$

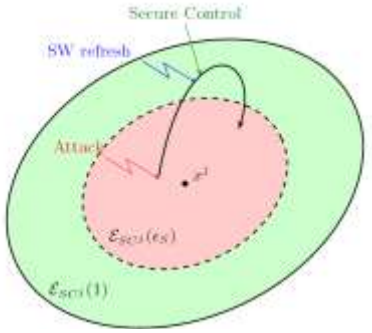
**Safety Set:** $\varepsilon_{SCj}(\epsilon_s) \triangleq \epsilon_s\, \varepsilon_{SCj}(1)$

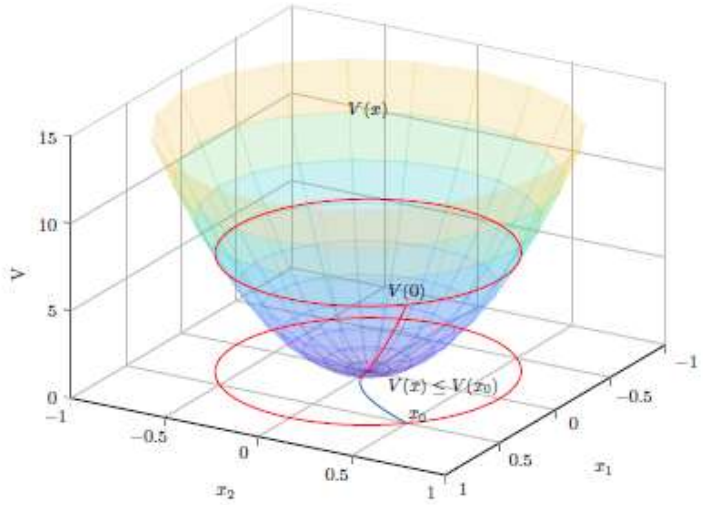
**Controlled System**: $\dot{x} = f_\varphi(x) \triangleq f(x, \varphi(x))$
**Lyapunov Function**: $V_\varphi : \mathbb{R}^n \to \mathbb{R}$, $\mathcal{N}_{V_\varphi}(x_{eq}) \subseteq \mathcal{N}_\varphi(x_{eq})$,
$V_\phi(x_{eq}) = 0$ and $\forall x \in \mathcal{N}_{V_\varphi}(x_{eq}) - \{x_{eq}\}$ : (i) $V_\varphi(x) > 0$,

$$\dot{V}_\varphi(x) = \frac{\partial V}{\partial x} \cdot f_\varphi(x) < 0$$

**Lyapunov level set**:For $\epsilon > 0$,

$$\mathcal{E}_\varphi(\epsilon) = \{x \in \mathcal{N}_{V_\varphi}(x_{eq}) | V_\varphi(x) \leq \epsilon\}. \qquad \epsilon \leq 1$$



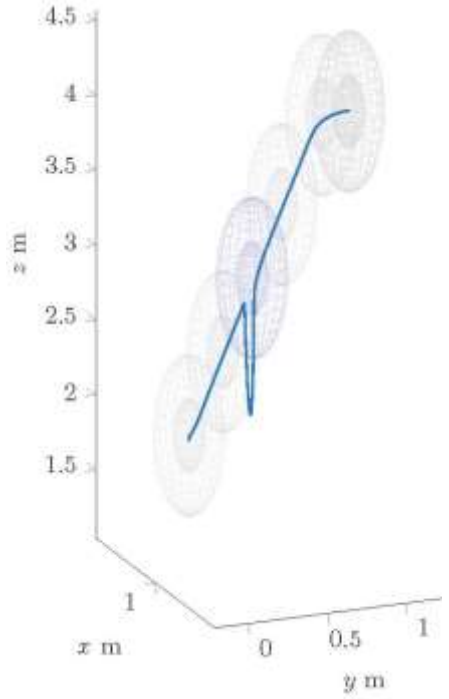




Raffaele Romagnoli, Bruce H. Krogh, Dionisio de Niz, Anton Hristozov, Bruno Sinopoli.
Software Rejuvenation for Safe Operation of Cyber-Physical Systems in the Presence of Runtime Cyber Attacks. IEEE Transactions on Control Systems Technology. to appear. 2023.

# Analysis of Mission Progress Enforcing Unsafe Behavior

**6 DOF ⇒ 12 state variables**

$$\ddot{p}_x = -\cos\phi\sin\theta\frac{F}{m}$$
$$\ddot{p}_y = \sin\phi\frac{F}{m}$$
$$\ddot{p}_z = g - \cos\phi\cos\theta\frac{F}{m}$$
$$\ddot{\phi} = \frac{1}{J_x}\tau_\phi$$
$$\ddot{\theta} = \frac{1}{J_y}\tau_\theta$$
$$\ddot{\psi} = \frac{1}{J_z}\tau_\psi.$$




**Linear design:**

- **linearize at equilibrium**
- **assume full state available**
- **LQ state feedback design**
- **reference points = equilibrium states**

# Drone Experiment

# Conclusions

"Who Builds a House Without Drawing Blueprints?" - Leslie Lamport

- Using the **mixed-trust framework** we can have safety guarantees for **complex systems**
- In systems where **not all parts are verified**, it is critical to **verify** the **interaction** between trusted and untrusted components
- **Modeling** of protocols **increases** our **assurance** in the correctness of the properties that we want to enforce

# Future Work

"Who Builds a House Without Drawing Blueprints?" - Leslie Lamport

- The model can help us improve the implementation

- How can we guarantee that the code follows the model?
  - Automatically generate C code that follows the UPPAAL model
    - Verify the translation

  - Verify that the current implementation follows the model
    - Prove the temporal properties at the code level (e.g., using Frama-C)