

You Can't Always Check What You Wanted:

Selective Checking and Trusted Execution to Prevent False Actuations in Real-Time Internet-of-Things

Monowar Hasan

School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA, USA
Email: monowar.hasan@wsu.edu

Sibin Mohan

Department of Computer Science
The George Washington University, Washington, DC, USA
Email: sibin.mohan@gwu.edu

Abstract—Modern Internet-of-Things devices are vulnerable to attacks targeting outgoing actuation commands that modify their physical behaviors. We present a “selective checking” mechanism that uses game-theoretic modeling to identify the suitable subset of commands to be checked in order to deter an adversary. This mechanism is coupled with a “delay-aware” trusted execution environment to ensure that only verified actuation commands are ever sent to the physical system, thus maintaining the safety and integrity of the system. Our proposed *selective checking and trusted execution (SCATE)* framework is implemented on an off-the-shelf ARM platform running embedded Linux and tested on four realistic IoT-specific cyber-physical systems (a ground rover, a flight controller, a robotic arm and an automated syringe pump).

I. INTRODUCTION

Many critical Internet-of-Things (IoT) systems (e.g., autonomous and self-driving cars, avionics, drones, power grids, medical devices, etc.) require tight conjoining of and coordination between computational and physical resources. These devices are being increasingly interconnected, often via the Internet, giving rise to the Real-Time Internet-of-things (RT-IoT) [1]. RT-IoT systems often have limited resources (processor, memory, battery life) and must meet stringent *timing* constraints. For instance, an industrial robot on a manufacturing line must carry out its operation (e.g., placing an object on a conveyor) in 50–100 ms [2]. Failure to do so could disrupt the entire manufacturing operation and even put the safety of the plant and human operators at risk! Modern IoT systems with such “real-time” requirements are increasingly becoming targets for cyber-attacks. The traditional approaches of air-gapping such systems or using proprietary protocols and hardware have been found wanting in the face of recent high-profile attacks [3]–[5].

In critical RT-IoT systems, *adversaries usually focus on destabilizing the system* [6], [7], oftentimes using the *falsification of actuation commands* — i.e., commands that control the state of the physical system are either modified or replaced while in transit to the physical component. In this work, we intend to *check actuation commands before they can affect the state of the physical system*. To prevent tampering, we *implement the checking mechanism in a trusted execution environment (TEE)* that is available on modern commodity processors, viz., the ARM TrustZone [8]. In

The material in this paper is based upon work supported in part by the U.S. National Science Foundation (NSF) under grant NSF 2246937. Any findings, opinions, recommendations or conclusions expressed in the paper are those of the authors and do not necessarily reflect the views of sponsors.

979-8-3503-3902-4/23/\$31.00 ©2023 IEEE

an ideal scenario, every outgoing actuation command should be checked. A serious hurdle that prevents such a strategy is that, as mentioned earlier, RT-IoT systems have stringent timing requirements — very often the actuation command, once sent out, *must be received by the physical system in a short, fixed, amount of time*. This limits the amount of time delays that can be introduced during the checking process. In addition, the control software has its own timing constraints, e.g., it must complete execution before a certain “deadline”; failure to do so can also cause instability in the system. Hence, we *cannot check (and thus, delay) every command* since each check compounds the delays faced by the system. Besides, TEEs introduce additional delays due to context-switch overheads, further (negatively) affecting the deadlines.

There is a need to carefully consider *how many* and *which* actuation commands are to be checked — to ensure that (a) the system safety/timing requirements are met and (b) also deter attackers. To this end, we *develop a mechanism to validate a (non-deterministic) subset of commands, varying at run-time* that significantly increases the difficulty for would-be attackers. We use a *game-theoretic formulation* of a two-player normal-form game [9]–[11] for the “selective checking” of the actuation commands (Sec. V). The combined framework is called the *selective checking and trusted environment (SCATE)* system. In this paper, we made the following contributions:

- We present a framework, SCATE, that protects IoT-specific real-time cyber-physical systems¹ from attacks that falsify actuation commands [Sec. III].
- We use a combination of game-theoretic analysis and a trusted environment to deter attackers, significantly reduce checking overheads, and still guarantee the safety and integrity of the RT-IoT systems [Sec. V].
- Our open-sourced implementation is available in a public repository [12].

II. BACKGROUND

We now provide a background the TEE technology (ARM TrustZone [8]) and the game-theoretic modeling [10]) used in our work.

1) *ARM TrustZone*: Trusted environments are set of hardware and software-based security extensions where the processors maintain a separated subsystem in addition to the traditional OS (also called rich OS) components. TEE technology has

¹In this paper we use the terms “*real-time IoT (RT-IoT)*” and “*cyber-physical systems (CPS)*” interchangeably.

been implemented on commercial hardware such as ARM TrustZone [8] and Intel SGX [13]. In this work, we consider TrustZone as the building block of our model due to the wide usage of ARM processors in IoT applications. We note that although we use the TrustZone functionality for demonstration purposes, our ideas are rather general and can be adapted to other TEE technology without loss of generality.

TrustZone contains two different privilege blocks: (a) regular (non-secure) execution environment, called “Normal World” (NW) and (b) a trusted environment, referred to as “Secure World” (SW). The NW is the untrusted environment that runs a commodity untrusted OS (called rich OS) whereas SW is a protected computing block that only runs privileged instructions. TrustZone hardware ensures that the resources in the SW can not be accessed from the NW. These two worlds are bridged via a software module, the *secure monitor*. The context switch between the NW and SW is performed via a *secure monitor call* (SMC).

2) *Normal-Form Games*: The overheads for the TEE context switch are costly (Sec. IV). If a task cannot verify all the actuation commands, we propose to select only a subset of commands in each job for checking. For this, we leverage the tools from game theory [14] to ensure that the chosen subsets are non-deterministic, at least from the adversary’s point of view (see Sec. V for details). In multi-agent systems, if the optimal action for one agent depends on the actions that the other agents take, game theory is used to analyze how an agent should behave in such settings. In a *normal-form game* [10], every player $j \in \{1, 2, \dots, J\}$ has a set of strategies (or actions) σ_j and a utility function $u_j: \sigma_1 \times \sigma_2 \times \dots \times \sigma_J \rightarrow \mathbb{R}$ that maps every outcome (a vector consisting of a strategy for every player) to a real number. As we shall see in Sec. V-A we formulate our problem as a *two-player game* (e.g., system designer and adversary). The output of the game finds the probability distribution over the player’s strategies (i.e., fraction of time a given strategy is selected in the game) that leads to an optimal outcome. While game-theoretic analysis has been used in other modeling problems (e.g., patrolling [11], network routing [15], transportation systems [16]) as well as general-purpose control systems/CPS [17]–[20], they are not real-time aware and do not consider the problem of protecting physical actuators.

III. MOTIVATION AND OVERVIEW

A. The Requirement for Actuation Checks

Without explicit control and verification over the actuation process, it is possible to send arbitrary signals to the actuators and an adversary can drive the system in undesirable ways. For instance, consider ground rovers that can be used in multiple cyber-physical applications such as remote surveillance, agriculture and manufacturing [21]. For demonstration purposes, we use a COTS ground rover running an embedded variant of Linux on an ARM Cortex-A53 platform (Raspberry Pi [22]).

We carried out a line-following mission where the rover steered from an initial location to a target location (Fig. 1a). A controller task runs the standard, pre-packaged, proportional–integral–derivative (PID) closed-loop control [23]. A 5-byte value is sent to the actuator (via memory-mapped registers)

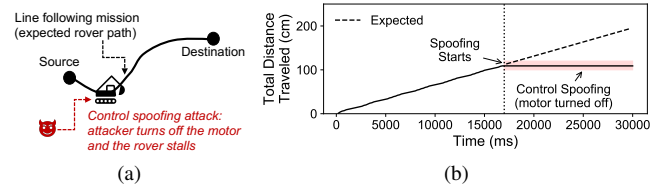


Fig. 1. Illustration of control spoofing attack: (a) schematic of our experiment setup; (b) readings from encoders under attack.

to control the wheel motors via the I2C interface [24]. The x-axis of Fig. 1 shows the time and the y-axis is the total distance traveled by the rover (i.e., readings from the wheel motors). Since the vendor implementation of the controller does not verify control commands, we were able to inject a logic bomb and send spoofed commands to turn off the motor (at the location marked in the figure). As a result, the rover deviates from its mission. The dashed line (after $t=17$ seconds) shows the expected behavior (viz., without any attack) as a reference (obtained by running a linear regression test). As the shaded region shows, the encoder readings remain the same and the rover was not following the line after the attack.

B. System Model

We consider a set of priority-driven, periodic real-time tasks, Γ , running on a multicore IoT platform Π . The set of tasks $\Gamma_p \subset \Gamma$ running on a given core $\pi_p \in \Pi$ is fixed and given by the designers. Each task τ_i issues N_i number of actuation requests.² We assume that there is a designer-given quality-of-service (QoS) requirement that $N_i^{min} \leq N_i$ actuation requests must be checked for each invocation of a task. We further assume that each actuation command a_i^j is associated with a designer-provided weight ω_i^j representing the importance/preference of checking the corresponding command over another. A higher weight implies that the actuation request is more critical, and designers want to examine it more often [25], [26].

C. Adversary Model

We assume that an adversary can tamper with the existing control logic to manipulate actuation commands, thus modifying the behavior of a system in undesirable ways (i.e., threaten the safety of the system). We only consider the cases where an adversary’s actions result in modifying actuation commands. Other classes of attacks such as scheduler side-channel attacks [27], [28], timing anomalies [29], [30] and network-level man-in-the-middle attacks [26], [31] are not within the scope of this work. We do not make any assumptions about how an adversary compromises tasks or actuation commands. For instance, bad software engineering practices leave vulnerabilities in the systems [32]. We note that embedded real-time CPS have fewer resources and lesser security protections, making some attacks easier for the adversary [7], [27], [29], [33]. We do not consider the adversarial cases that require physical access, i.e., the attacker can not physically control/turn off/damage the actuators or the system.

²Note: RT-IoT systems are deterministic due to their stringent timing/safety requirements. Hence, actuation command sequences are predefined at the design time. These commands are well known, carefully characterized, and do not change at runtime.

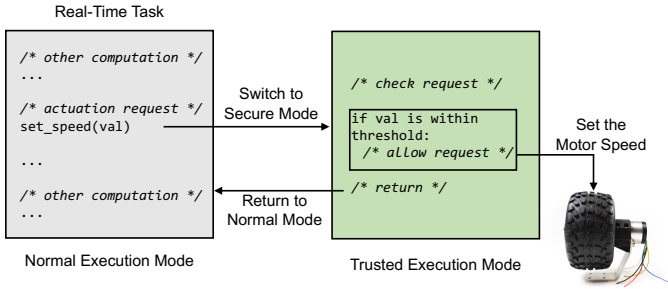


Fig. 2. Flow of operation in SCATE.

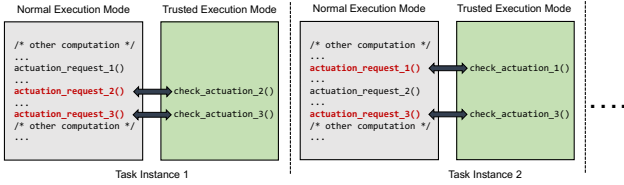


Fig. 3. Non-deterministic selection of actuation commands.

D. Problem Overview and Our Approach

Actuation commands that are malicious can jeopardize the safety and integrity of cyber-physical applications. In this research, we propose techniques to *protect systems from control spoofing attacks by examining actuation commands before* they are issued to physical peripherals. We name our mechanism, SCATE (selective checking and trusted execution) where we consider cyber-physical applications consisting of software tasks that can have two different types of execution sections: (a) regular (potentially untrusted) execution section where normal executions are carried out and (b) trusted sections where critical information (*i.e.*, actuation events) are examined.

Note: One may wonder why we cannot execute all parts of the real-time tasks (*i.e.*, regular execution and actuation checks) within secure enclaves. This can be problematic since, (a) it will increase the size of the trusted computing base (TCB), (b) secure enclaves may become exposed to potentially insecure, external interfaces (*e.g.*, networks), (c) any vulnerabilities that exist in the real-time codebase will be carried into the TEE and (d) the TEEs may not have required library support and will require significant engineering effort to execute real-time tasks, thus increasing the possibility of introducing further vulnerabilities and bugs. In general, since the real-time control software requires access to external interfaces (I/O, network, *etc.*) for correct operation it exposes more attack surfaces. Moving the checking mechanism to the TEE *and keeping it as small as possible* ensures that while adversaries can compromise the main software and control system, they will be unable to bypass SCATE— thus ensuring that the main system is protected from malicious behavior. This is similar to the methods employed in SGX (and similar mechanisms) [34]–[36] where the entire application (or even the operating system) does not execute inside the secure enclaves; only security-critical code (and data) executes inside the enclaves.

The high-level schematic of SCATE is depicted in Fig. 2. When a task issues an actuation command, SCATE transfers the control

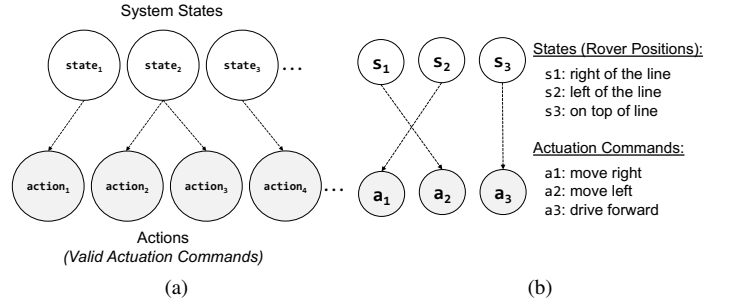


Fig. 4. (a) $State \rightarrow Action$ mapping used in SCATE. (b) Example states and valid actuation commands for a line-following rover.

to the secure mode using the TrustZone SMC instructions.³ In the secure mode, a designer-provided trusted entity checks the actuation commands. The context switch overhead for switching between normal and trusted modes is negligible. For example, consider the rover used in our experiments (Sec. VI). The execution frequency of the controller task is 5 Hz (200 ms), and it generates four actuation commands (to set the speeds and direction of attached motors). The controller task must complete execution before its periodic invocation interval (200 ms). If we check the speed and direction values of each of the four commands the controller task fails to comply with its timing requirements (since it requires 261 ms to finish). For such situations SCATE *selectively checks a subset of the actuation events* (Sec. V). Figure 3 shows an illustrative case where a task generates three actuation requests and we can check at most two requests to comply with timing/safety requirements. In this case SCATE randomly checks two commands each time. From the earlier rover example, by reducing the number of checks by half (*e.g.*, randomly checking two commands in each task instance instead of all four), the controller task in SCATE manages to finish before 200 ms without significantly degrading security as explained in Sec. VI.

IV. CHECKING ACTUATION COMMANDS

In SCATE, a “checking module” executes inside the trusted environment. The checking module observes system states and decides whether a given actuation command is legitimate or malicious. Our scheme is similar to the broad class of “rule-based checking” mechanisms that are used for intrusion detection in a wide variety of domains, *e.g.*, Linux security modules [38], network routers [39], software security auditing [40], cloud environments [41], to name a few.

For a given RT-IoT platform we assume that there exists a $CheckAct(\tau_i, a_i^j, t)$ function⁴ that examines a given actuation command a_i^j (where $1 \leq j \leq N_i$, N_i is the total number of commands the task issues) generated by a task τ_i at a given time t . As shown in Fig. 4a, the checking module uses policy abstraction rules [42], *viz.*, $State \rightarrow Action$ pairs where the $State$ predicate represents a given system state and $Action$ denotes corresponding valid actuation command(s). In particular, we assume that when τ_i executes an actuation command, function $CheckAct(\tau_i,$

³Our current implementation uses OP-TEE TEEC_InvokeCommand() API [37] to perform the switching (see Sec. VI).

⁴The exact function depends on the specific CPS and application requirements.

a_i^j, t) first observes the system state $\mathcal{S}(t)$ and then decides whether the actuation command a_i^j is valid for the current state $\mathcal{S}(t)$. Consider the line-following rover presented earlier. The directions for the wheels of the rover (*i.e.*, forward, left and right; controlled by the attached motors) are the actuation commands. At any given point in time, the rover can be in one of three states: $\mathcal{S} = \{\text{ON_LINE}, \text{LEFT_OF_LINE}, \text{RIGHT_OF_LINE}\}$ that denotes whether the rover is on top of the line or shifted left/right of the line, respectively.

The rover controller task performs the following actuation operations (*i.e.*, actions): `move_left()/move_right()` (move the rover left/right, respectively) and `move_forward()` (drive the rover forward). This rover’s corresponding *State* \rightarrow *Action* mapping is illustrated in Fig. 4b. If the rover is on the left side of the line (*i.e.*, *State* = LEFT_OF_LINE), the valid command should be `move_right()` (*i.e.*, shift the rover back so that it stays on the line) and if the rover is on top the line (*i.e.*, *State* = ON_LINE), the controller task should drive it forward (*i.e.*, issue `move_forward()` command).

We note that the ideas presented here are agnostic to the specific checking method and SCATE is compatible with existing techniques (*e.g.*, defining rules at design times [43], [44], deriving from specifications [45] and based on statistical analysis [46]).

A. Requirements for Coarse-Grain Checking

In order to check actuation commands we must ensure that SCATE should not cause inordinate delays and the timing requirements of real-time tasks are satisfied (*i.e.*, they complete execution before their respective deadlines). Hence we develop design-time tests (see Appendix A) that ensure tasks meet their timing requirements (deadlines). Researchers [47]–[49] show that although TEEs are implemented on hardware, they can still cause significant overheads — this is particularly acute in real-time applications. For instance, consider the Linux-based TrustZone port, OP-TEE [37] supported on many embedded platforms. Our experiments show that the overhead of switching between normal to trusted mode is around 66 ms for OP-TEE on a Raspberry Pi platform. For completeness, we also performed experiments on an ARMv8-M Cortex-M33 architecture using ARM FVP libraries [50] where the regular applications were running on FreeRTOS [51] and trusted mode codes were executed on bare-metal. We find that the mode switching delays in this setup are 2 ms. The delays are higher in the Linux environment due to extra overheads (*i.e.*, execution of a sequence of API calls [47]) imposed by the Linux kernel and OP-TEE secure OS. Although the overheads of secure calls (SMC) for switching between regular and trusted modes are platform-specific, it may still not be feasible to check multiple commands while meeting real-time guarantees. For example, if a task operates at 50 Hz (*i.e.*, it is required to finish before 20 ms) [27] and regular computation takes 10 ms, the FreeRTOS-based setup allows at most 5 checks in order to comply with timing requirements. Likewise, for the applications running on a Linux and OP-TEE-based Raspberry Pi platform (Sec. VI-B), we can check at most 3 commands per instance if the controller task operates at 5 Hz. We, therefore, need smart techniques, say where *only a subset of commands are vetted*

while maintaining security guarantees, to support both security and performance for real-time applications. We now present our methods to achieve this based on game-theoretic analysis.

V. GAME-THEORETIC ANALYSIS

Let us consider the case when there exists a task τ_i such that it cannot perform all the N_i checks before its deadline (denoted by D_i). One option to reduce the number of checks is to verify *only a subset of commands* so that the task can finish before its deadline.⁵ That is, check a subset of commands, K_i , ($N_i^{\text{min}} \leq K_i < N_i$) such that $R_i^{\text{TEE}} \leq D_i, \forall \tau_i \in \Gamma$ where R_i^{TEE} is the response time (*i.e.*, the time between task arrival to completion, used to verify that the task meets its deadline [53]). The maximum time for checking K_i commands is known at design time. The challenge then is to decide *which* subset of K_i (among N_i) actuation requests should be selected for checking in each task τ_i . In addition, if we use a “deterministic selective checking” approach and examine only a fixed set of K_i commands and an adversary jeopardizes some or all of the remaining $N_i - K_i$ requests, then the attack will succeed and remain undetected. To balance the security and real-time requirements, SCATE *selects different subsets of requests for checking each time the task executes*. In particular, during each task execution, we pick a set of K_i commands with pre-computed probability distributions. While we pick a subset of commands, it should look like (to the adversary) that SCATE *is checking all commands*. We formulate this problem as a two-player game [10] and develop Algorithm 1 to determine the feasible number of K_i inspection points that provide a similar level of security when compared to the case that checks all N_i requests (see Sec. V-A). Our game model considers *all possible adversarial actions* and ensures that *all commands are eventually checked* within a time window. This is different than arbitrarily random selection, where some commands may not be checked at all.

Intuition and Example. Let us consider a ground rover. The rover controller task (τ_c) generates the following actuation requests ($N_c = 3$): (a) `setEncL(val)` and `setEncR(val)` that set the speed of the left and right motor encoders, respectively (denoted by a_c^1 and a_c^2); (b) `setNav(cmd)` that issues a navigation command where each *cmd* specifies values to the peripheral registers for navigating the rover forward, backward, left or right directions (denoted by a_c^3). The weights given by: $\Omega_c = \{\omega_c^1, \omega_c^2, \omega_c^3\}$. The weights are used to determine which commands should be checked more often. For example, if $\omega_c^3 = 2$ and $\omega_c^1 = \omega_c^2 = 1$, then SCATE tends to check `setNav(cmd)` twice as often as the other two commands. The checking for a_c^1 and a_c^2 is whether the speed value is within a given bound (*e.g.*, $val \in [v^-, v^+]$) and for a_c^3 the checking module verifies if the *cmd* value is consistent so that the rover is on the line and is correctly following the mission.

We now consider the case when checking all three requests does not comply with the timing requirement of task τ_c and we can only verify at most $K_i = 2$ requests. Therefore, the

⁵Another alternative could be to check all (or some) commands at longer time scales. However, this approach is vulnerable to TOCTTOU (time of check to time of use) attacks [52] and may result in delayed detection. Hence we do not consider this alternative in our model.

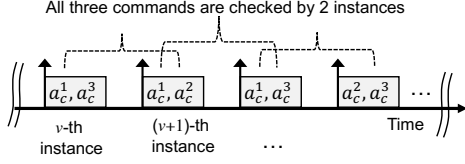


Fig. 5. Non-deterministic checking: *all* commands are eventually checked.

possible combinations for checking are as follows: $X_c = \{\{a_c^1, a_c^2\}, \{a_c^2, a_c^3\}, \{a_c^1, a_c^3\}\}$. In SCATE, for each instance of the task τ_c we need to select *any* j -th element from the set X_c with probability x_c^j that provides better “monitoring coverage.” Hence, we generate a random number between (0,1), check if it is less than x_c^j , and pick the corresponding element from X_c . Eventually, we cycle through *all* of the elements of X_c , thus ensuring, (a) an attacker cannot rely on a predetermined set of checks and (b) all the commands are eventually checked in a fixed time window. For example, let $x_c^1 = x_c^2 = 0.25$ and $x_c^3 = 0.5$. Then, for any given instance of τ_c the possibility of verifying both a_c^1 and a_c^2 is 25%, verifying both a_c^2 and a_c^3 is 25%, where the possibility of verifying a_c^1 and a_c^3 is 50% (recall that by assumption we can check only 2 commands per job). Figure 5 presents an instance of such execution. SCATE ensures that although we pick a subset of commands each time, eventually *all* the commands are checked. For example, in Fig. 5 SCATE requires two instances to check all three commands. In the following section, we present our ideas to compute these probabilities using game-theoretic analysis.

A. Generating Non-Deterministic Schedules

We model the selection of a subset of commands (for checking) as a *two-player normal-form game* [9]–[11]. This is non-deterministic since the subset of commands checked by SCATE is not known a priori. Each instance of the task checks a different subset of commands and all the commands are ultimately checked (see Fig. 5).

1) *Game Setup*: We consider two players: the leader (*i.e.*, system designers) and the follower (*i.e.*, adversary). Let X_i denote the set of all combinations of choosing K_i subset of commands from total N_i number of possibilities, *i.e.*, the size of set $|X_i| = \binom{N_i}{K_i} = \frac{N_i!}{K_i!(N_i-K_i)!}$. In game-theoretic terminology, X_i represents the set of leader’s “strategies.” Let us now introduce the variable Q_i that represents the attacker’s set of actions (*i.e.*, follower’s strategies). The set Q_i represents the possible combinations of actuation requests invoked by task τ_i that can be compromised by an adversary. Recall from the rover example where the controller task τ_c invokes $N_c = 3$ actuation requests, hence, the adversary can pick one of the following eight combinations: $Q_c = \{\{a_c^1\}, \{a_c^2\}, \{a_c^3\}, \{a_c^1, a_c^2\}, \{a_c^2, a_c^3\}, \{a_c^1, a_c^3\}, \{a_c^1, a_c^2, a_c^3\}, \{\emptyset\}\}$. For example, the first element in the set denotes the adversary chooses to compromise only invocation a_c^1 , the fifth element implies both a_c^2 and a_c^3 are compromised while the last element implies there is no attack during this instance of the task. The size of the attacker’s strategy set Q_i is 2^{N_i} .

Recall that each actuation command a_c^j is associated with a designer-given weight ω_c^j . A higher weight implies that designers want to check the corresponding command more often. For

System Reward:

$$\lambda_i^{j,l} = \frac{\sum_{w \in \Psi(X_i^j)} w}{\sum_{w \in \Psi(X_i^j) \cup \Psi(Q_i^l)} w}, \quad (1)$$

normalization factor: union of designer’s and adversary’s strategies

System Cost:

$$\zeta_i^{j,l} = \frac{\sum_{w \in \Psi(Q_i^l)} w}{\sum_{w \in \Psi(X_i^j) \cup \Psi(Q_i^l)} w}, \quad (2)$$

normalization factor: union of designer’s and adversary’s strategies

Fig. 6. Reward and cost functions.

instance, from the rover example, designers may want to check navigation commands (a_c^3) more frequently than the ones that set the wheel speeds (a_c^1, a_c^2) and may set higher weight for ω_c^3 . Let $\Lambda(X_i^j)$ denote the set of commands used for vetting, and $\Psi(X_i^j)$ is the set of corresponding weights in the j -th element of the strategy set X_i . Likewise, $\Lambda(Q_i^l)$ denotes the set of commands, and $\Psi(Q_i^l)$ is the corresponding set of weights compromised by the attacker in its l -th strategy. In the rover example, if we select $j=2$ and $l=4$ (*i.e.*, second and fourth elements of the designers and adversary’s strategy set) then $\Lambda(X_c^2) = \{a_c^2, a_c^3\}$, $\Psi(X_c^2) = \{\omega_c^2, \omega_c^3\}$ and $\Lambda(Q_c^4) = \{a_c^1, a_c^2\}$, $\Psi(Q_c^4) = \{\omega_c^1, \omega_c^2\}$.

We now introduce two variables, *viz.*, *system reward* (λ) and *system cost* (ζ). A higher system reward and lower cost are good for the designers and bad for the attackers. Likewise, higher system costs and lower rewards are favorable from the adversary’s point of view (and bad for the designers). If a task τ_i selects the j -th element from the set of strategies X_i and the attacker selects the l -th strategy from Q_i for attack then the system reward is $\lambda_i^{j,l}$ and the cost is $\zeta_i^{j,l}$. If the task selects a subset of commands for vetting in its j -th strategy and the adversary also attacks those invocations in its l -th strategy, *i.e.*, $\Lambda(X_i^j) \cap \Lambda(Q_i^l) \neq \emptyset$, it implies that the attack is detected. Hence, we set $\lambda_i^{j,l}$ to a large positive value (*i.e.*, high system reward, since the attack is detected) and $\zeta_i^{j,l}$ a large negative value (*i.e.*, no system cost). In contrast, if $\Lambda(X_i^j) \cap \Lambda(Q_i^l) = \emptyset$ for any pair (j,l) , *i.e.*, $\Lambda(X_i^j)$ does not contain any commands in attacker’s l -th strategy $\Lambda(Q_i^l)$, that implies the compromised commands are not vetted (*i.e.*, the spoofed command is not checked). In this case, we set $\lambda_i^{j,l}$ a large negative value (*i.e.*, no reward) and $\zeta_i^{j,l}$ a large positive value (*i.e.*, high system cost). When the above two conditions do not hold (*i.e.*, only a subset of the compromised commands are checked) and therefore $\exists(j,l)$ such that $\Lambda(X_i^j) \cap \Lambda(Q_i^l) \neq \emptyset$, we then obtain the system reward/cost by *normalizing the weights* of both adversary and designer’s strategies. For this, we define the reward and cost functions in Eq. (1) and Eq. (2), respectively

Algorithm 1 SCATE: Parameter Selection

Input: Input taskset parameters Γ
Output: For each task τ_i , the size of the feasible set $K_i^* \geq N_i^{min}$ and selection probability $x_i^j, j=1, \dots, |X_i^*|$ for each of the combinations in the strategy set X_i^* ; Infeasible otherwise.

- 1: /* Check minimum feasibility requirements */
- 2: **for each** $\tau_i \in \Gamma$ **do**
- 3: Set $K_i = N_i^{min}$ and calculate response time R_i^{TEE} (see Appendix A)
- 4: **end for**
- 5: **if** $\exists \tau_i$ such that $R_i^{TEE} > D_i$ **then**
- 6: **return** Infeasible /* Unable to integrate SCATE for given QoS requirements */
- 7: **end if**
- 8: **for each** task τ_i (from higher to lower priority order) **do**
- 9: Find maximum $K_i^* \in [N_i^{min}, N_i]$ using logarithmic search such that all low-priority tasks τ_l meet their timing requirements (i.e., $R_l^{TEE} \leq D_l$)
- 10: /* Obtain parameters for non-deterministic checking */
- 11: **if** $K_i^* < N_i$ **then**
- 12: Determine the strategy set X_i^* for K_i^* where $|X_i^*| = \binom{N_i}{K_i^*}$ and obtain probabilities x_i^j by solving the game formulation
- 13: **end if**
- 14: Update response time R_i^{TEE} for each τ_l that executes with a priority lower than τ_i with the updated size K_i^*
- 15: **end for**
- 16: **return** the size of the feasible set K_i^* and probability x_i^j of selecting j -th strategy ($j=1, 2, \dots, |X_i^*|$) from X_i^* for each task $\tau_i \in \Gamma$

(see Fig. 6). Let us revisit the rover example with $j=2$ and $l=4$. In this case $\lambda_c^{2,4} = \frac{\omega_c^2 + \omega_c^3}{\omega_1^1 + \omega_2^2 + \omega_3^3}$ and $\zeta_c^{2,4} = \frac{\omega_c^1 + \omega_c^2}{\omega_1^1 + \omega_2^2 + \omega_3^3}$. These reward and cost functions give us one way to measure the security of the system in terms of how many significant actuation commands we can monitor given an attacker's strategy. A higher system reward (and lower cost) implies that SCATE performs more checking with respect to a given adversarial action.

2) *Formulation as an Optimization Problem:* Let us now denote x_i^j as the probability of selecting the j -th element from X_i (represents the proportion of times in which a strategy j is used by the task τ_i in the game). The output of the game will provide the probability distribution of (non-deterministically) selecting a subset of K_i commands from the set possible choices (i.e., X_i) for the different execution instances of a given task τ_i . For a given adversarial strategy l , summing over all the strategy sets X_i (i.e., $\sum_{j=1}^{|X_i|} x_i^j \lambda_i^{j,l}$ and $\sum_{j=1}^{|X_i|} x_i^j \zeta_i^{j,l}$) gives us the total system reward and cost, respectively. We can obtain probability distributions of selecting elements from X_i for a given attacker strategy l (that maximizes the system reward) by forming a linear optimization program. For each of the attacker's l -th strategy ($1 \leq l \leq |Q_i|$), we compute a strategy for the task τ_i such that (i) playing l -th strategy is the best response from the adversary's point of view (i.e., more system cost) and (ii) under this constraint, the strategy maximizes the reward for τ_i (i.e., checks critical commands more often). Appendix B presents the details of our linear programming formulation.

B. Calculating Feasible Command Set

The game formulation from the previous sections assumes that we know the size of the set K_i and calculate the probabilities accordingly. However, in a system with multiple real-time tasks, finding the size of feasible set K_i for each task $\tau_i \in \Gamma$ while also meeting the real-time requirements (deadlines) is a non-trivial problem. Hence, we develop an iterative solution for finding the size of this set (see Algorithm 1). In Lines 1–4, we first assign $K_i = N_i^{min}, \forall \tau_i$ and check whether all tasks meet their timing requirements (i.e., finish before their deadlines). If there exists a

TABLE I
SUMMARY OF OUR IMPLEMENTATION PLATFORM

Artifact	Configuration
Platform	Broadcom BCM2837 (Raspberry Pi 3)
Hardware	1.2 GHz 64-bit Cortex-A53, 1 GB memory
Operating system	Linux (rich OS), OP-TEE (TEE)
Kernel version	Linux kernel 4.16.56, OP-TEE core 3.4
Interface	I2C
Boot parameters	dtparam=i2c_arm=on, dtparam=spi=on, force_turbo=1, arm_freq=1200, arm_freq_min=1200, arm_freq_max=1200

task that fails to meet its timing requirements, we report that it is “infeasible” to integrate SCATE in the target system while still satisfying designer-specified QoS requirements (Line 7). This infeasibility result provides hints to the designers to either update or modify system parameters to enable the ability to check actuation commands in the system. Otherwise, we optimize the number of commands a task can verify in an iterative manner (Lines 9–16). To be specific, for a given task τ_s we perform a logarithmic search (see Algorithm 2 in Appendix C) and find the maximum number of commands K_i^* . If the selected parameter K_i^* is less than the total commands N_i , we then use game theoretical analysis from Sec. V-A and obtain probabilities of selecting different sequences (Line 13). The above process is repeated for all the tasks.

VI. EVALUATION





A. Implementation

We implemented SCATE on Raspberry Pi 3 Model B [22]. We used the Adafruit motor shield [54] (an I/O extension board) that allowed us to control multiple actuators using the I2C interface. We used an open-source motor driver [55] and servo controllers [56]. We implemented the trusted execution modes using the OP-TEE [37] software stack. We used an Ubuntu 18.04 filesystem with a 64-bit Linux kernel (version 4.16.56) as the rich OS and executed our CheckAct() functions in the OP-TEE secure kernel (version 3.4). We note that our implementation using Raspberry Pi, Linux and OP-TEE serves as a good proof-of-concept and can be extended with other OS, hardware platforms and TEE architectures without loss of generality. Our implementation code is available in a public repository [12]. Table I summarizes the system configurations and implementation details.

B. Experiments with Cyber-Physical Platforms

We chose four realistic real-time cyber-physical platforms as case-studies to evaluate the efficacy of SCATE: (a) ground rover, (b) UAV flight controller, (c) robotic arm and (d) syringe/infusion pump that are used in many cyber-physical applications. These are *off-the-shelf platforms and we did not modify them*. Table II summarizes the properties of each of these systems and attack/detection techniques used in our experiments. Unlike generic applications, there are few publicly available open-source real-time platforms due to their proprietary nature. In addition, there is a significant amount of effort involved in setting up a TEE-supported real-time cyber-physical platform and generating evaluation traces from it. We, therefore, limit ourselves to four

TABLE II
REAL-TIME IOT PLATFORMS USED IN OUR EXPERIMENTS

Platform	Application	Real-Time Requirements	Actuation Commands	Attack Demonstration	Checks inside Enclave
	The rover performs a line following mission. The controller task sets the speed of the rover and steers the wheels (based on its position on the line) by executing a PID control loop	Set the speed and direction of the motors for the wheel movements within sampling interval (<i>i.e.</i> , control loop frequency, set at 5 Hz)	<ul style="list-style-type: none"> Set the speed of the wheels Set wheel directions (left, right, forward and backward) 	DoS attack [44]: arbitrarily sets high speed for one of the wheel motors	The speed of motors can only be within predefined limit
	Executes a PID control loop and issues PWM signals to four motors connected to the four propellers of a quad-copter	Issue the PWM signal within sampling frequency interval (5 Hz in our setup) to ensure the quad-copter is stable	<ul style="list-style-type: none"> Set PWM frequency Set PWM pulse duration (four, one for each of the propellers) 	Parameter corruption attack [7]: modify PID control coefficients and send incorrect PWM pulse to the front right motor	Check PID control coefficients (<i>i.e.</i> , pulse duration values) before issuing PWM signals to the motos
	The robot arm performs the following operations in a sequence: pick an object (close it claws), move the arm to destination position, drop the object (open claws) and reset the arm back to initial position	Complete movement of the object before arrival of the next object; inter-arrival duration of the objects was set at 250 ms	Set rotation angle for each of the four servos	Synchronization attack [57]: sends incorrect <code>angle</code> value to the servo channel and prevents the arm from resetting back to its initial position	Check the consistency of each (<code>channel</code> , <code>angle</code>) pair (<i>i.e.</i> , the <code>angle</code> value for a given servo <code>channel</code> can not be more than the designer provided bounds) before issuing pulses to the servo motors
	The pump pushes certain amount of fluid and then pulls the trigger to reset the syringe to its initial position	Perform the push/pull operations within designer specified time limit (set at 300 ms)	<ul style="list-style-type: none"> Set motor rotation frequency Drive the motor forward/backward to push the fluids and reset the motor position 	Bolus tampering attack [57], [58]: the attacker injects more fluid than the permitted volume	Checks the amount of fluid the motors can pump (<i>i.e.</i> , monitor the number of PUSH/PULL events the controller task invokes)

real-time platforms in this paper — albeit they cover a wide range of application domains and should suffice to demonstrate the feasibility of our approach. We assume equal weights for actuation commands. We use fault injection [57], [59] to mimic malicious behavior and trigger attacks that are known to the checking module (*i.e.*, `CheckAct()` function). Note that this is a standard technique used by the researchers to evaluate security solutions in cyber-physical applications [7], [21], [44], [57], [60], [61].

• **Case-Study #1 (Ground Rover):** Our first case-study platform is the ground rover introduced earlier. We used an open-source implementation of the rover controller (written in Python) [62] and ported it to C for compatibility with OP-TEE APIs. The rover performed a line-following mission where it moved from a source to a target way-point by following a line. Each instance of the controller task first set the speed of the motor for the wheel movements and then steered (*e.g.*, forward, left or right) based on its position on the line.

Actuation: The rover has four actuation commands: two for setting the speed of both of the wheels and two for issuing navigation commands to the two motors attached to the wheel.

Attack and Consequences: We injected a DoS attack [44] that arbitrarily sets a high speed for one of the motors (to destabilize the rover and move it away from the line). This attack can deviate the rover from its way-points (or even crash it) due to the imbalance in the wheel speeds.

Detection: In our setup, the `CheckAct()` functions validate whether the rover speed is within designer-given predefined thresholds [63] and also verify whether the navigation commands were consistent with the rover position. We detected the DoS attack by checking the bounds on the speed (*i.e.*, 70–100 decimal values [63]) issued by the controller task.

• **Case-Study #2 (Flight Controller):** Our second case-study is

a flight-controller for a quad-copter [64]. The original controller code is developed for Arduino platforms. Since Arduino boards do not support TrustZone, we ported it to the Raspberry Pi and OP-TEE-enabled environments. In this setup the controller executes a PID control loop using the Ziegler–Nichols method [65] and sends pulse width modulation (PWM) signals to spin each of four motors (*i.e.*, actuators) connected to the propellers.

Actuation: There are five actuation commands: one for setting the PWM frequency and the other four are for sending PWM pulse durations for each of the motors to rotate the copter propellers. The `CheckAct()` functions validate whether the PWM frequency and pulse durations sent to each of the motors were within a certain range (obtained from PID control logic).

Attack and Consequences: For this case study we considered a parameter corruption attack [7] that modifies the control parameters (*e.g.*, the PID control coefficients) at runtime and sends incorrect pulse values to the front-right motors. This attack can suddenly turn off/freeze the propellers. As a result, the copter will instantly fall/crash.

Detection: This attack is detected since we verify the PID parameters and corresponding PWM pulse durations.

• **Case-Study #3 (Robotic Arm):** Our next case-study platform is a robotic arm used in manufacturing systems. The movement of the robotic arm is controlled by four servos (actuators in our context). Each servo is connected to a specific “channel” (I/O port) in the Adafruit motor shield. We use an open-source Python-based robot controller [66] and adapted the implementation for our C-based setup.

Actuation: Our robot performed an assembly line sequence with the following four actuation operations: `PICK()`, `MOVE()`, `DROP()` and `RESET()` that (*i*) picks an object from the first position, (*ii*) moves the arm to a final position, (*iii*) drops the

object and, finally, (iv) resets the arm to initial position (to pick up another object). Each operation takes a (channel, angle) pair that controls the rotation of the corresponding servo to the desired angle (45° in our setup [66]).

Attack and Consequences: We used a synchronization attack [57] that destabilizes the assembly line by preventing the robot from resetting its arm back to the initial position. To demonstrate this, we injected a logic bomb that sets an incorrect angle value in the RESET() operation (e.g., servo channel 3). This attack can collapse the whole assembly line since the arm is not returned to the initial position and hence is unable to pick up objects queued in the line.

Detection: CheckAct() detects this attack since it asserts that each servo can only move up to a certain designer-provided angle (45°) for each of the operations.

• **Case-Study #4 (Syringe Pump):** Our final case-study platform is a syringe/infusion pump [67]. In our experiments, we considered a bolus delivery use-case [58], [68] where the syringe pump first pushes a certain amount of fluid (PUSH event) and then pulls the trigger back (PULL event). The syringe movement is controlled by a stepper motor. Since the original implementation is for Arduino platforms (and does not support TrustZone), we modified the codes to make it compatible with Raspberry Pi and its motor driver library.

Actuation: For a given fluid amount, the syringe pump implementation selects the number of steps where the PUSH and PULL events should be called. We considered each of the PUSH/PULL events as actuation requests since they set the direction of motor rotation. In our setup, there were seven actuation requests: one for setting the motor rotation frequency and six for PUSH and PULL events (three each). PULL events were called after all three PUSH operations were completed.

Attack and Consequences: We implemented a bolus tampering attack [57], [58] where the adversary injects more fluid than is required (i.e., more than three PUSH events). The attack has serious safety consequences and is a health hazard since it can inject more fluids/medications into the patient body than the permitted amount.

Detection: CheckAct() verifies the motor frequency and how many times each of the PUSH/PULL events are called. When more than three events are triggered for a given run, we flag this as an attack.

1) **Experience and Findings:** We compare SCATE against a naive scheme [44] that checks *all* the actuation commands; we refer to that technique as the “fine-grain” checking scheme. The goal of our evaluation was to *study the trade-offs between security and real-time requirements*. We, therefore, considered the subset of commands selected for checking was no more than 50% of the total number of commands (i.e., $K = \lfloor 0.5N \rfloor$) so that tasks can finish before their timing requirements.⁶ Our experiments address the following research questions (RQs):

- **RQ1.** How quickly an intrusion can be detected by SCATE when compared to the fine-grain scheme [44]?

⁶We also carried out additional experiments to show the effect of varying this parameter (see Sec. VI-C).

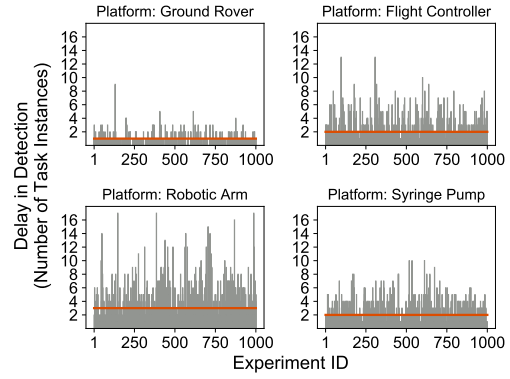


Fig. 7. Delay (viz., number of instances) in detecting an intrusion.

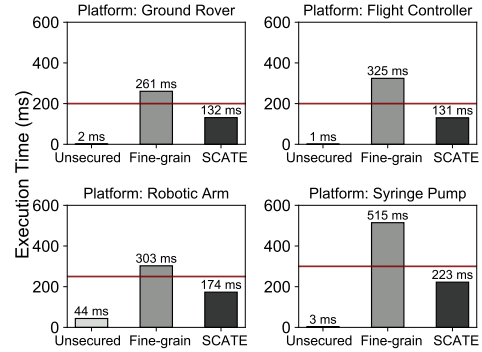


Fig. 8. Execution time of the controller task.

- **RQ2.** What are the performance impacts and runtime overheads of these schemes?

Security Analysis. We use “detection delay” as a security metric. In the first set of experiments (Fig. 7) we analyze the delay in detecting the attacks: a comparison between SCATE and the fine-grain scheme. For a given platform and for each of our experiments (x-axis of Fig. 7) we triggered the attack randomly during the execution of the victim task and measured the time delay (in terms of the number of task instances⁷, y-axis in Fig. 7) In the fine-grain scheme, the time to detect an attack is upper bounded by the sampling interval (period), T_i (i.e., requires at most one task instance). From our experiments, with 1000 individual trials for each of the four platforms, we found that the mean and 99th-percentile detection delay were 1–3 and 3–12 sampling intervals, respectively (refer to Table III for exact values). We note that this delay in detection results in improved response time and reduced resource usage (see more in the following experiments).

SCATE can provide similar levels of security when compared to fine-grained checking since, on average, it requires only 1–3 additional task instances to detect the attacks.

Timeliness and Overhead Analysis. We also compare with traces from *vanilla* execution scenarios when there is no verification of actuation commands (i.e., tasks are always running in the rich OS). We refer to this vanilla execution

⁷If the detection delay is \hat{y} instances, it implies that SCATE requires $\hat{y}T_c$ additional time units to detect the attack compared to the fine-grained checking where T_c is the period.

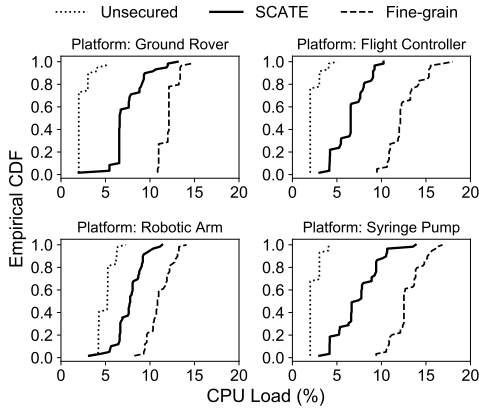


Fig. 9. Empirical CDF of the load. SCATE uses 30.48%–47.32% less CPU.

as “unsecured” since it does not protect the system from any adversarial actions. Figure 8 (y-axis) shows the execution times (captured using the Linux `clock_gettime()` function and the `CLOCK_PROCESS_CPUTIME_ID` clock). The horizontal line represents the deadlines, *i.e.*, if the response time of the task is above the margin, the task misses its deadline and the physical system will become unstable (and hence unsafe). Both the fine-grained checking and SCATE increase response times when compared to the unsecured scheme since there is no context switch between Linux and OP-TEE in the latter. Note that the increase in computing resources due to the integration of additional security checking/protection techniques (*e.g.*, cryptographic operations, intrusion detection) is an expected side-effect to improve security [7], [33], [47], [69]–[71]. The fine-grained scheme expends more time since it verifies all N actuation commands, *i.e.*, there are more context switches (from rich OS to secure enclave) and runtime checking overheads. As a result, the controller task easily misses its deadline and drives the system into an unsafe state for all of our case-studies. In contrast, SCATE allows the tasks to finish within deadlines.

Fine-grained checking increases execution times and controller tasks fail to comply with their timing requirements. SCATE, by design, manages to complete execution before deadlines.

Figure 9 shows the resource usage. For this, (a) we executed the controller tasks independently for 60 seconds, (b) observed the CPU load using the `/proc/stat` interface and (c) report the results from 1000 individual trials. The x-axes of Fig. 9 show the CPU load and y-axes show the corresponding cumulative distribution function (CDF). From our experiments we found that SCATE increases CPU usage by 1.5–3.2 times when compared to an unsecured scheme — this is expected since vanilla execution does not provide any security guarantees (and there is no additional context switch overhead). We also note that SCATE *reduces CPU load by 30.48%–47.32%* when compared to the fine-grain scheme; this could be useful for many applications (say for battery-operated systems to improve thermal efficiency). Table III summarizes our findings.

In comparison with fine-grained checking, on average, SCATE uses 30.48%–47.32% less CPU for its operations.

TABLE III
COMPARISON WITH FINE-GRAIN CHECKING: SUMMARY

Platform	Performance Metrics: SCATE vs Fine-grain			
	Ex. Time Reduction (%)	CPU Use Reduction (%)	Delay (Task Instances)	
			Mean	99 th -p
Ground Rover	49.69	37.58	1	3
Flight Controller	59.81	47.32	2	8
Robotic Arm	42.80	30.48	3	12
Syringe Pump	56.81	42.87	2	8

C. Simulation-based Evaluation

We also developed a simulator [12] and conducted experiments with randomly generated workloads for a broader design-space exploration. In Fig. 10 we present the trade-off between real-time and security guarantees. For this, we introduce a metric called “coverage ratio” (CR). The CR metric shows us how many actuation commands (out of the total number of commands) we can check without violating timing constraints. We define CR as follows: $CR = \frac{1}{|\Gamma|} \sum_{\tau_i \in \Gamma} \frac{K_i}{N_i}$, where $\frac{1}{|\Gamma|} \sum_{\tau_i \in \Gamma} \frac{N_i^{min}}{N_i} \leq CR \leq 1$, $|\Gamma|$ is the total number of tasks and the parameter K_i is obtained from Algorithm 1. We further use the “schedulability” metric — a useful mathematical tool developed by the real-time community to analyze whether all activities of a given system can meet their timing constraints even in the worst-case behavior of the system [72]. A given taskset is considered as *schedulable* if *all* the tasks in the taskset meet their timing requirements (*i.e.*, response time is less than or equal to the deadline).

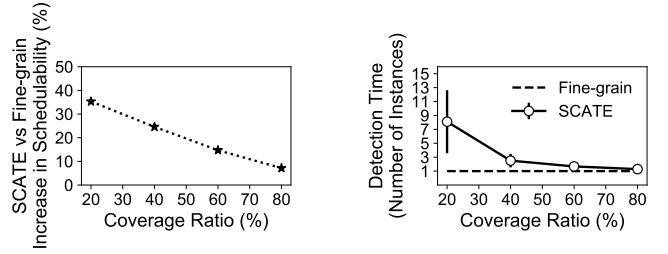


Fig. 10. Real-time vs security trade-offs: low coverage ratio, while increasing the acceptance ratio (left), can lead to an increase in detection time (right).

The x-axis of Fig. 10 shows the coverage ratio. The y-axis in the left figure shows the increase in schedulability in SCATE when compared to the fine-grain scheme while the right figure shows the detection time. Figure 10 shows that there is a trade-off between real-time and security requirements: a lower coverage ratio increases the schedulability (since there are lower checking overheads) but increases the detection times. This is because if the coverage ratio is low, only a few commands are selected for checking during each instance and a vulnerable/compromised command will only be verified infrequently; thus resulting in longer detection times.

*If we perform fewer checks in each task instance, we can accommodate more tasks in the system (*i.e.*, higher acceptance ratio). However, this may result in delayed detections (*e.g.*, on average, requires eight additional task instances).*

Summary. Our experiments reveal interesting trade-offs between real-time and security requirements. Fine-grain checking — while providing better security guarantees (*i.e.*, lower detection time) — can negatively affect schedulability and, hence, the safety and integrity of the system. SCATE, in contrast, provides better schedulability guarantees but may result in slower detection times. By using our approach, designers of the systems can now customize their platforms and selectively verify actuation commands based on application requirements.

VII. DISCUSSION

In this work, we do not consider the aftereffects of detecting an intrusion. Our current implementation blocks malicious commands. Other strategies could involve the raising of alarms and/or sending out buffered (or even predetermined) alternate commands. SCATE transfers the control to the secure enclave for a (non-deterministic) subset of actuation requests. Although this does not jeopardize the safety of the physical system (see more in the following paragraph), an adversary can still send spoofed signals in lieu of unchecked commands (perhaps in a brute-force manner). Alternative design choices (to further improve security and minimize overhead) could be: (*i*) pass all the actuation requests through the trusted enclave but checks only a few commands or (*ii*) buffer multiple commands together and then transfer control *once* to the enclave for checking (*i.e.*, check them in a batch). However, this may cause additional delays in serving actuation requests. We intend to incorporate these features and study trade-offs in future work.

SCATE can protect against cases where an adversary injects new commands or blocks existing commands (*i.e.*, DoS attacks) by checking whether the *number of commands generated during each execution window matches the design-time specifications*. While SCATE adds delays in detection (*e.g.*, on average 1–3 task instances when compared to the scheme that checks all the actuation commands, see Table III), we can still retain the safety and normal operations of the plant due to physical inertia.

In this paper we assume the existence of “perfect” checking modules provided by the system engineers (*i.e.*, an attack is always correctly detected). Depending on the implementation, CheckAct() functions may result in false-positive/false-negative errors. Our model can also handle such cases by incorporating the detection-inefficiency factors in calculating reward/cost metrics. For example, if the detection accuracy of CheckAct() is 95%, one way to express reward and cost functions is as follows: $\lambda_{\text{Imperfect}} = (1 - .05)\lambda$ and $\zeta_{\text{Imperfect}} = (1 + .05)\zeta$, respectively.

VIII. RELATED WORK

Our earlier research [44] introduced a naive version of checking actuation commands; that preliminary (workshop) paper *does not provide any analytical guarantees* and checks *all* actuation commands. In contrast, we propose a novel game-theoretical analysis and “selectively” (and non-deterministically) check a subset of commands that *guarantees* that all system requirements (*e.g.*, timing constraints) are met. We compare SCATE with prior work (referred to as fine-grain scheme) and find that naively checking all commands results in missed

deadlines (see Sec. VI-B). As we see in our experiments, checking all commands results in high overheads and tasks fail to comply with their timing requirements. In contrast, the novel game-theoretical model used in SCATE retains real-time guarantees while providing a similar level of security. In addition, we have implemented the solution on an actual ARM+Linux platform and evaluated it against *multiple realistic CPS platforms*.

Perhaps the closest line of work to ours is PROTC [36] uses a monitor in the enclave enforces secure access control policy for some peripherals of the drone and ensures that only authorized applications can access certain peripherals. Unlike our scheme, PROTC [36] is limited for specific applications (*i.e.*, aerial robotic vehicles), requires a centralized control center to validate/enforce security policies, and does not consider any load balancing issues. In other work [73]–[76], we proposed mechanisms to secure legacy systems by integrating additional, independent, periodic monitoring tasks. They are pure software-based solutions and do not guarantee the integrity of security checks like SCATE. In contrast, we now consider a TEE-based system where security checks (*i.e.*, actuation command verification using a trusted enclave) are interleaved within real-time tasks. There exists other work (Crystal [77] and M2Mon [78]) that address actuators attacks. Unlike SCATE, they are (*a*) limited for aerial systems and (*b*) do not consider temporal constraints. There exist various hardware/software-based mechanisms and architectural frameworks [6], [60], [79]–[83]. However, they are not designed to protect against control-specific attacks and may not be suitable for systems developed with COTS components. Existing CPS anomaly detection approaches [7], [21], [84]–[86] also do not consider real-time aspects. Researchers use game-theoretical analysis for (*a*) control systems [17], [19], (*b*) decision making problems [18] and (*c*) preventing physical intrusions in CPS [20]. These schemes are not designed to protect the systems against false actuation commands. In addition, they are not timing-aware (*i.e.*, real-time requirements are not considered). There also exists a large number of research for generic IoT/cyber-physical systems (see related surveys [8], [87]–[90]) — however, the consideration of actuation-specific security scheduling and overload management aspects of real-time applications distinguish our work from other research. To the best of our knowledge, this is the first comprehensive work that introduces the notion of randomized coarse-grain checking for overloaded systems in order to validate actuation commands in a TEE-enabled real-time CPS.

IX. CONCLUSION

In this paper we present a framework, SCATE, to enhance the security and safety of the time-critical IoT systems. We use a combination of trusted hardware and the intrinsic real-time nature of such systems and propose techniques to selectively verify a subset of commands that provides a trade-offs between real-time and security guarantees. We believe that our technique can be incorporated into multiple IoT-specific application domains such as avionics, automobiles, industrial systems, medical devices, unmanned and autonomous vehicles.

REFERENCES

- [1] C.-Y. Chen et al. Securing real-time Internet-of-things. *Sensors*, 18(12), 2018.
- [2] K. Castelli et al. Development of a practical tool for designing multi-robot systems in pick-and-place applications. *MDPI Robotics*, 8(3), 2019.
- [3] N. Falliere et al. W32. stuxnet dossier. *White paper, Symantec Corp.*, 5:6, 2011.
- [4] S. S. Clark and K. Fu. Recent results in computer security for medical devices. In *MobiHealth*, pp. 111–118, 2011.
- [5] S. Checkoway et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Sec. Symp.*, 2011.
- [6] F. Abdi et al. Preserving physical safety under cyber attacks. *IEEE IoT J.*, 6(4):6285–6300, 2018.
- [7] H. Choi et al. Detecting attacks against robotic vehicles: A control invariant approach. In *ACM CCS*, pp. 801–816, 2018.
- [8] S. Pinto and N. Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM CSUR*, 51(6):130, 2019.
- [9] J. C. Harsanyi and R. Selten. A generalized nash solution for two-person bargaining games with incomplete information. *INFORMS Man. Sci.*, 18(5-part-2):80–106, 1972.
- [10] V. Conitzer and T. Sandholm. Computing the optimal strategy to commit to. In *ACM EC*, pp. 82–90, 2006.
- [11] P. Paruchuri et al. An efficient heuristic approach for security against multiple adversaries. In *IFAAMAS AAMAS*, pp. 1–8, 2007.
- [12] SCATE implementation. https://github.com/mnwrhms/scate_implementation.
- [13] V. Costan and S. Devadas. Intel SGX Explained. *IACR Crypt. ePrint Arch.*, (086):1–118, 2016.
- [14] T. Roughgarden. Algorithmic game theory. *Comm. of the ACM*, 53(7):78–86, 2010.
- [15] Y. A. Korilis et al. Achieving network optima using Stackelberg routing strategies. *IEEE/ACM TON*, 5(1):161–173, 1997.
- [16] J. Cardinal et al. Pricing of geometric transportation networks. In *CCCG*, pp. 92–96, 2005.
- [17] S. Moothedath et al. A game-theoretic approach for dynamic information flow tracking to detect multi-stage advanced persistent threats. *IEEE TACON*, 2020.
- [18] G. Yang et al. Adaptive learning in two-player stackelberg games with continuous action sets. In *IEEE CDC*, pp. 6905–6911, 2019.
- [19] J. Chen and Q. Zhu. A game-theoretic framework for resilient and distributed generation control of renewable energies in microgrids. *IEEE Trans. on Smart Grid*, 8(1):285–295, 2016.
- [20] S. Rass et al. Physical intrusion games—optimizing surveillance by simulation and game theory. *IEEE Access*, 5:8394–8407, 2017.
- [21] P. Guo et al. RoboADS: Anomaly detection against sensor and actuator misbehaviors in mobile robots. In *IEEE/IFIP DSN*, pp. 574–585, 2018.
- [22] Raspberry Pi. <https://tinyurl.com/rpi3modelb>.
- [23] Dexter Industries Sensors. https://github.com/DexterInd/DI_Sensors.
- [24] I²C manual, 2003.
- [25] S. Di Leonardi et al. Maximizing the security level of real-time software while preserving temporal constraints. *IEEE Access*, 2023.
- [26] V. Lesi et al. Security-aware scheduling of embedded control tasks. *ACM TECS*, 16:188:1–188:21, 2017.
- [27] C.-Y. Chen et al. A novel side-channel in real-time schedulers. In *IEEE RTAS*, pp. 90–102, 2019.
- [28] S. Liu et al. Leaking your engine speed by spectrum analysis of real-time scheduling sequences. *J. of Sys. Arch.*, 97:455–466, 2019.
- [29] R. Mahfouzi et al. Butterfly attack: Adversarial manipulation of temporal properties of cyber-physical systems. In *IEEE RTSS*, pp. 93–106, 2019.
- [30] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE RTAS*, pp. 357–367, 2019.
- [31] V. Lesi et al. Network scheduling for secure cyber-physical systems. In *IEEE RTSS*, pp. 45–55, 2017.
- [32] F. Loi et al. Systematically evaluating security and privacy for consumer IoT devices. In *ACM IoTS&P*, pp. 1–6, 2017.
- [33] C. H. Kim et al. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.
- [34] J. Wang et al. S-Blocks: Lightweight and trusted virtual security function with SGX. *IEEE Trans. on Cloud Comp.*, pp. 1–1, 2020.
- [35] D. Goltzsche et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *IEEE/IFIP DSN*, pp. 386–397, 2018.
- [36] R. Liu and M. Srivastava. PROTC: PROTeCting drone’s peripherals through ARM trustzone. In *ACM Dronet*, pp. 1–6, 2017.
- [37] Open Portable Trusted Execution Environment. <https://www.op-tee.org/>.
- [38] B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *ACM/IFIP WITS*, pp. 1–12, 2004.
- [39] L. Popa et al. Building extensible networks with rule-based forwarding. In *USENIX OSDI*, pp. 379–392, 2010.
- [40] C. Jang et al. Rule-based auditing system for software security assurance. In *IEEE ICUFN*, pp. 198–202, 2009.
- [41] R. Rajendran et al. Detection of dos attacks in cloud networks using intelligent rule based classification system. *Springer Cluster Comp.*, 22(1):423–434, 2019.
- [42] T. Yu et al. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-things. In *ACM HotNets*, pp. 1–7, 2015.
- [43] S. Adepu and A. Mathur. From design to invariants: Detecting attacks on cyber physical systems. In *IEEE QRS-C*, pp. 533–540, 2017.
- [44] M. Hasan and S. Mohan. Protecting actuators in safety critical IoT systems from control spoofing attacks. In *ACM IoT S&P*, pp. 8–14, 2019.
- [45] R. Berthier and W. H. Sanders. Specification-based intrusion detection for advanced metering infrastructures. In *IEEE PRDC*, pp. 184–193. IEEE, 2011.
- [46] V. Chandola et al. Anomaly detection: A survey. *ACM CSUR*, 41(3):15, 2009.
- [47] A. Mukherjee et al. Optimized trusted execution for hard real-time applications on cots processors. In *ACM RTNS*, pp. 50–60, 2019.
- [48] J. Amacher and V. Schiavoni. On the performance of arm trustzone. In *IFIP DAIS*, pp. 133–151, 2019.
- [49] Y. Liu et al. RT-trust: Automated refactoring for trusted execution under real-time constraints. In *ACM GPCE*, pp. 175–187, 2018.
- [50] ARM Fixed Virtual Platforms. <https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms>.
- [51] FreeRTOS. <http://www.freertos.org>.
- [52] J. Wei and C. Pu. Toctou vulnerabilities in unix-style file systems: An anatomical study. In *USENIX FAST*, volume 5, pp. 12–12, 2005.
- [53] N. Audsley et al. Applying new scheduling theory to static priority pre-emptive scheduling. *SE Journal*, 8(5):284–292, 1993.
- [54] Adafruit motor shield. <https://learn.adafruit.com/adafruit-motor-shield>.
- [55] Adafruit driver. https://github.com/threebrooks/AdafruitStepperMotorHAT_CPP.
- [56] PCA9685 I2C PWM driver. <https://github.com/TeraHz/PCA9685>.
- [57] M. R. Aliabadi et al. Artinali: dynamic invariant detection for cyber-physical system security. In *ACM ESEC/FSE*, pp. 349–361, 2017.
- [58] L. Cheng et al. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *ACM ACSAC*, pp. 315–326, 2017.
- [59] F. M. Tabrizi and K. Pattabiraman. Flexible intrusion detection systems for memory-constrained embedded systems. In *IEEE EDCC*, pp. 1–12, 2015.
- [60] M.-K. Yoon et al. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *IEEE RTAS*, pp. 21–32, 2013.
- [61] F. Abdi et al. Guaranteed physical security with restart-based design for cyber-physical systems. In *ACM/IEEE ICCPS*, pp. 10–21, 2018.
- [62] Raspberry Pi rover. <https://github.com/Veilkrand/simplePiRover>.
- [63] GoPiGo. <https://github.com/DexterInd/GoPiGo>.
- [64] Drone controller. <https://github.com/lobodol/drone-flight-controller>.
- [65] K. J. Åström and T. Häggglund. Revisiting the Ziegler–Nichols step response method for PID control. *Elsevier J. of proc. con.*, 14(6):635–650, 2004.
- [66] Robot arm control. <https://github.com/tutRPI/6DOF-Robot-Arm>.
- [67] C-FLAT implementation. <https://github.com/control-flow-attestation/c-flat>.
- [68] T. Abera et al. C-FLAT: control-flow attestation for embedded systems software. In *ACM CCS*, pp. 743–754, 2016.
- [69] T. Xie and X. Qin. Improving security for periodic tasks in embedded systems through scheduling. *ACM TECS*, 6(3):20, 2007.
- [70] S. Mohan et al. Real-time systems security through scheduler constraints. In *Euromicro ECRTS*, pp. 129–140, 2014.
- [71] R. Pellizzoni et al. A generalized model for preventing information leakage in hard real-time systems. In *IEEE RTAS*, pp. 271–282, 2015.
- [72] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM CSUR*, 43(4):35:1–35:44, 2011.
- [73] M. Hasan et al. Contego: An adaptive framework for integrating security tasks in real-time systems. In *Euromicro ECRTS*, pp. 23:1–23:22, 2017.
- [74] M. Hasan et al. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *IEEE RTSS*, pp. 123–134, 2016.
- [75] M. Hasan et al. A design-space exploration for allocating security tasks in multicore real-time systems. In *DATE*, pp. 225–230, 2018.
- [76] M. Hasan et al. Period adaptation for continuous security monitoring in multicore systems. In *DATE*, 2020.

- [77] S. Etigowni et al. Crystal (ball) i look at physics and predict control flow! just-ahead-of-time controller recovery. In *ACM ACSAC*, pp. 553–565, 2018.
- [78] A. Khan et al. {M2MON}: Building an {MMIO-based} security reference monitor for unmanned vehicles. In *USENIX Security*, pp. 285–302, 2021.
- [79] S. Mohan et al. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM HiCoNS*, pp. 65–74. ACM, 2013.
- [80] M.-K. Yoon et al. Memory heat map: anomaly detection in real-time embedded systems using memory behavior. In *ACM/EDAC/IEEE DAC*, pp. 1–6, 2015.
- [81] M.-K. Yoon et al. Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In *ACM/IEEE IoTDI*, pp. 191–196, 2017.
- [82] F. Abdi et al. ReSecure: A restart-based security protocol for tightly actuated hard real-time systems. In *IEEE CERTS*, pp. 47–54, 2016.
- [83] D. Lo et al. Slack-aware opportunistic monitoring for real-time systems. In *IEEE RTAS*, pp. 203–214, 2014.
- [84] F. Fei et al. Cross-layer retrofitting of UAVs against cyber-physical attacks. In *IEEE ICRA*, pp. 550–557, 2018.
- [85] S. McLaughlin. CPS: stateful policy enforcement for control system device usage. In *ACM ACSAC*, pp. 109–118, 2013.
- [86] R. Lanotte et al. Runtime enforcement for control system security. In *IEEE CSF*, pp. 246–261, 2020.
- [87] A. Humayed et al. Cyber-physical systems security – A survey. *IEEE IoT J.*, 4(6):1802–1831, 2017.
- [88] Y. Yang et al. A survey on security and privacy issues in Internet-of-Things. *IEEE IoT J.*, 4(5):1250–1258, 2017.
- [89] M. Ammar et al. Internet of Things: A survey on the security of IoT frameworks. *Elsevier J. of Inf. Sec. & App.*, 38:8–27, 2018.
- [90] W. Li et al. Research on ARM TrustZone. *ACM GetMobile*, 22(3):17–22, 2019.
- [91] J. Chen. Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks. In *Euromicro ECRTS*, pp. 251–261, 2016.

APPENDIX

A. Feasibility Conditions

Let N_i be the number of actuation requests generated by task τ_i that require vetting, T_i is the inter-arrival time (*i.e.*, period). Further, let C_i^o be an upper bound of additional computing time required due to (a) context switching (from normal execution to secure enclave and returning the context back to normal mode) and (b) performing the checks inside the enclave. Then, the execution time of τ_i can be represented as $C_i^{TEE} = C_i + N_i C_i^o$. The task τ_i is “schedulable” if the worst-case response time (WCRT), R_i^{TEE} , is less than deadline, *i.e.*, $R_i^{TEE} \leq D_i$. We can calculate an upper bound of R_i^{TEE} using traditional response-time analysis [91] as follows: $R_i^{TEE} = C_i^{TEE} + \sum_{\tau_h \in hp(\tau_i, \pi_p)} \left(1 + \frac{D_i}{T_h}\right) C_h^{TEE}$, where $hp(\tau_i, \pi_p) \in \Gamma_p$ denotes the set of tasks that are higher-priority than τ_i running on core π_p . For a feasible system, *all* tasks must be schedulable, *viz.*, $R_i^{TEE} \leq D_i, \forall \tau_i \in \Gamma$. Let $R_i = C_i + \sum_{\tau_h \in hp(\tau_i, \pi_p)} \left(1 + \frac{D_i}{T_h}\right) C_h$ denote the vanilla response time (*i.e.*, when there is no checking). Notice that the task τ_i will miss its deadline if $R_i^{TEE} > D_i$. We can deduce that τ_i will miss its deadline if the following condition holds: $O_i > D_i - R_i$, where $O_i = N_i C_i^o + \sum_{\tau_h \in hp(\tau_i, \pi_p)} \left(1 + \frac{D_i}{T_h}\right) N_i C_h^o$ is the total overhead for checking the actuation commands including context switching in/out of the secure enclave.

B. Linear Programming Formulation for solving the Game

We can obtain the probability distributions for selecting the elements from X_i (*i.e.*, the set of all combinations of choosing

Algorithm 2 Calculation of Maximum Feasible Checks

```

1: Define  $K_i^l := N_i^{min}$ ,  $K_i^r := N_i$ ,  $K_i^c := 0$ 
2: Set  $\widehat{K}_i := \{N_i^{min}\}$  /* Initialize a variable to store feasible values */
3: while  $K_i^l \leq K_i^r$  do
4:   Update  $K_i^c := \lfloor \frac{K_i^l + K_i^r}{2} \rfloor$ 
5:   if  $\exists \tau_l \in lp(\tau_i, \pi_p)$  such that  $\tau_l$  is not schedulable with  $K_i = K_i^c$  then
6:     /* Decrease verification load to make the taskset schedulable */
7:     Update  $K_i^r := K_i^c - 1$ 
8:   else
9:     /* Taskset is schedulable with  $K_i^c$  */
10:     $\widehat{K}_i := \widehat{K}_i \cup \{K_i^c\}$  /* Add  $K_i^c$  to the feasible list */
11:    /* Check schedulability with larger  $K_i$  for next iteration */
12:    Update  $K_i^l := K_i^c + 1$ 
13:  end if
14: end while
15: return  $\max(\widehat{K}_i)$  /* return the maximum from the set of feasible values */

```

K_i requests from total N_i actuation commands) for a given attacker strategy l (that maximizes the system reward) by forming the following linear program:

$$\max_{x_i^j} \sum_{j=1}^{|X_i|} x_i^j \lambda_i^{j,l} \quad (3a)$$

$$\text{s.t.} \quad \forall l' \in [1, |Q_i|], \sum_{j=1}^{|X_i|} x_i^j \zeta_i^{j,l'} \geq \sum_{j=1}^{|X_i|} x_i^j \zeta_i^{j,l'} \quad (3b)$$

$$\sum_{j=1}^{|X_i|} x_i^j = 1 \quad (3c)$$

$$x_i^j > 0, \forall j \in [1, |X_i|] \quad (3d)$$

The objective function in Eq. (3a) maximizes the total system reward. The constraint in Eq. (3b) ensures that the current (*e.g.*, l -th) strategy results in higher cost for the attacker when compared to other adversarial strategies. The constraint in Eq. (3c) ensures the sum of probability distributions equal to unity and the last constraint in Eq. (3d) ensures non-zero probabilities so that all combinations of the actuation commands from X_i can be selected.

Let $[x_i^j]_{j=1:|X_i|}(l)$ denote the solution obtained from the linear programming formulation for the l -th adversarial strategy. Then, from all feasible strategies l (where $1 \leq l \leq |Q_i|$) we choose the one (say l^*) that maximizes the objective value in Eq. (3a), *i.e.*, $l^* = \operatorname{argmax}_{1 \leq l \leq |Q_i|} \sum_{j=1}^{|X_i|} x_i^j \lambda_i^{j,l}$. The variables $[x_i^j]_{j=1:|X_i|}(l^*)$ obtained by solving the corresponding linear program gives us the probability distributions of selecting K_i subset of commands from a total of N_i commands. The game-theoretical analysis shows that the probability distributions obtained by solving the l^* -th linear program will be optimal for the task τ_i (*i.e.*, maximizes system reward) [10], [11].

For a given strategy l , the above linear programming formulation can be solved in polynomial time. Since the strategy set Q_i is finite by definition, we can calculate the optimal probability distributions (*e.g.*, $[x_i^j]_{j=1:|X_i|}(l^*)$) in a finite amount of time since the time is polynomial in the total number of adversarial strategies.

C. Calculation of Maximum Feasible Number of Commands

The pseudocode for calculating the maximum number of commands (K_i) that can be checked per job while guaranteeing feasibility is presented in Algorithm 2.