$\mathrm{ACAC_{ABCD}}$ Model: Implementation and Comparative Performance Study

Tanjila Mawla[1,*], Lopamudra Praharaj[1], James Benson[2], and Maanak Gupta[1]
[1]Tennessee Tech University, Cookeville, Tennessee, USA
[2]University of Texas at San Antonio, San Antonio, Texas, USA
tmawla42@tntech.edu, lpraharaj42@tntech.edu, james.benson@utsa.edu, mgupta@tntech.edu
*corresponding author

2025 11th International Symposium on System Security, Safety, and Reliability (ISSSR) | 979-8-3315-0124-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ISSSR65654.2025.00049

Abstract—The rapid advancements in networking, communication, automation engineering, and artificial intelligence (AI) are shaping society into a highly interconnected and intelligent ecosystem. In smart environments such as smart farming and manufacturing, devices autonomously perform long-lived operations with minimal human intervention, adapting dynamically to system requirements and environmental conditions. To regulate these long-lived operations, referred to as activities (e.g., spraying water, plowing field), an Activity-Centric Access Control (ACAC) model has been proposed. While previous works have incrementally introduced ACAC's decision parameters Authorizations (A), Obligations (B), Conditions (C), and Dependencies (D) and presented policy specifications and proof-of-concept implementation for dependencies (D), a comprehensive implementation of the consolidated $\mathrm{ACAC_{ABCD}}$ model remains unexplored. Validating ACAC's feasibility in highly dynamic environments is essential, particularly where security and safety must be ensured to protect both device operations and the system from potential insider and external threats. Additionally, the absence of constraints across various life-cycle stages undermines the model's comprehensiveness. In this paper, we take a significant step toward implementing the consolidated $\mathrm{ACAC_{ABCD}}$ model in a smart farming use case, incorporating additional constraints such as usage count, cardinality, dynamic separation of duties, and activity duration. We first provided the formal model definitions for authorizations (A), obligations (B) and conditions (C), referred to as $\mathrm{ACAC_A}$, $\mathrm{ACAC_B}$, and $\mathrm{ACAC_C}$ respectively. Furthermore, to demonstrate proof-of-concept, we utilize AWS cloud infrastructure. The Activity Control Decision ($\mathrm{ACD_{ACAC_{ABCD}}}$) policy engine is implemented using AWS Lambda, with AWS S3 storing policies and real-time data. An AWS EC2 instance manages interactions with Lambda, processing requests that specify a source and activity. IoT devices are configured via AWS IoT Core Things to dynamically reflect activity state changes. We simulated the $\mathrm{ACAC_{ABCD}}$ model on AWS, conducted a performance analysis based on multiple metrics and compared ACAC's efficiency against the widely used Attribute-Based Access Control (ABAC) model. Our results demonstrate that the ACAC model is practical, scalable, and well-suited for dynamic smart systems where device operations are interconnected and continuously evolving.

Keywords–Long-lived operations, Activity-Centric Access Control (ACAC), implementation, smart and connected systems, Amazon Web Service (AWS), policy

1. Introduction

The rapid evolution of smart and interconnected systems has enabled Internet of Things (IoT) devices capable of performing operations with minimal human intervention. Using advanced technologies such as embedded sensors, real-time applications, and computational methods, these systems analyze and respond to environmental data to enable automation, optimization, and enhanced functionalities in diverse domains such as smart homes, agriculture, transportation, and healthcare. However, the intricate and interconnected nature of these systems, governed by execution order, parallelism, environmental conditions, and potential conflicts, presents significant challenges in integration, security, and policy enforcement.

To address these challenges, access control mechanisms have played a crucial role in safeguarding resources from unauthorized access without compromising the safety and functionality of the system. With the increasing attack vectors and the automation of IoT-based cyber-physical systems (CPSs), it is essential to continuously evaluate system-related information, adapt to dynamically changing parameters, and regulate user access to resources as key aspects of security analysis. Traditional access control models, including Discretionary (DAC) [1], Mandatory (MAC) [2], and Role-Based (RBAC) [3], along with the more fine-grained Attribute-Based Access Control (ABAC) [4], provide fundamental access control capabilities in IoT-based CPS environments. However, these models are not inherently designed to handle the distributed and interconnected nature of resource-sharing systems. To address the need for real-time, context-aware security management, the Activity-Centric Access Control (ACAC) model was introduced which offers a dynamic approach to regulating long-lived device operations within these ecosystems [5], [6]. The devices' long continuous operations are referred to as activities. Mawla et al. proposed the access control decision components (Authorizations (A), Obligations (B), Conditions (C) and Dependencies (D) between activities) along with the continuous activity control mechanisms within an activity life cycle [6], [7]. Building on the discussion of the model components, Mawla et al. introduced a formally grounded mathematical model, $\mathrm{ACAC_D}$, to define activity dependencies (D) [7], along with a corresponding policy specification framework [8]. Furthermore, to enhance activity control policies, additional constraints such as usage count, object cardinality, dynamic separation of activities, and activity duration were incorporated. To formally express these constraints, a

2835-2823/25/$31.00 ©2025 IEEE
DOI 10.1109/ISSSR65654.2025.00049

constraint specification language was proposed using Object Constraint Language (OCL), ensuring precise and structured policy enforcement [9].

While the model development and policy specification for the dependencies between activities provide a strong ground to evaluate the viability of the proposed ACAC model, it is imperative to demonstrate its practical feasibility, security effectiveness, and scalability in real-world CPS domains. Implementation is essential to empirically validate the model's ability to enforce access policies efficiently, evaluate its performance under varying system loads, and assess its integration with existing security frameworks. Moreover, the existing works on ACAC model lack the analysis of the model by evaluating all decision parameters (A, B, C, D) which undermines the effectiveness of the model in real-world implementation.

In this paper, we take steps towards designing and maturing the comprehensive $\mathrm{ACAC_{ABCD}}$ model by incorporating all activity decision parameters (A,B,C,D). We first provide the model definitions for the authorizations, obligations and conditions adapting the attribute-based access control (ABAC) model and expressing the policies using Backus Naur Form (BNF). Later, we provide an Activity Control Decision ($\mathrm{ACD_{ACAC_{ABCD}}}$) algorithm to demonstrate flow of activity control and the evaluation of each parameter in the required phase of the activity cycle. This paper enforces the proposed ACAC model with all of its parameter A, B, C, D (referred as a consolidated $\mathrm{ACAC_{ABCD}}$ model) and the specified constraints [9] by following the ACD algorithm. To validate proof of concept and assess the model's performance, we implemented ACAC model using comprehensive use-case scenarios, demonstrating its practicality and efficiency through a comparative performance analysis. We utilize Amazon Web Services (AWS) to manage policies using AWS S3, implement the policy engine through AWS Lambda, and handle activity requests via EC2 virtual instances. During the simulation of use-case scenario, we demonstrate how devices dynamically update their status based on activity decisions, leveraging AWS IoT Core things to showcase the practicality of the ACAC model.

The rest of this paper is organized as follows. Section 2 presents the relevant background. Section 3 defines the policy evaluation approaches for Authorizations ($\mathrm{ACAC_A}$), Obligations ($\mathrm{ACAC_B}$), and Conditions ($\mathrm{ACAC_C}$) models. Section 4 outlines the algorithmic flow for evaluating all parameters throughout an activity's life cycle. In Section 5, we demonstrate a proof-of-concept implementation of the $\mathrm{ACAC_{ABCD}}$ model, covering system architecture, use-case scenarios, and performance evaluations. Finally, Section 6 presents the conclusion of the paper.

2. Relevant Background

The rapid development of technology and the expansion of IoT devices contribute to an increasing likelihood of security vulnerabilities. Several studies [10], [11] have explored security and privacy challenges in smart and interconnected systems. Additionally, extensive research has been conducted on access control models for IoT systems, as discussed in [12]–[17]. Existing access control models primarily focus on securing systems against potential attacks using various approaches, including Attribute-Based Access Control (ABAC) [15], [16], hybrid models integrating ABAC and Role-Based Access Control (RBAC) [18], Context-Aware Access Control [14], and Relationship-Based Access Control (ReBAC) [19]. While ABAC [4] is one of the most flexible and widely adopted access control models, it lacks mechanisms for managing long-lived operations, dependencies between those operations, and real-time access control based on dynamically checking and updating the dependencies. Thus, a critical gap remains in addressing the long-lived operations of IoT devices and the dependencies between those operations. Traditional access control models (DAC [1], MAC [2], RBAC [3]) focus on static authorization criteria but fail to account for the dynamic, ongoing nature of IoT operations that require real-time access decision management. While some frameworks, such as Usage Control (UCON) [20], explore ongoing access control for system resources, no existing work specifically addresses IoT access decisions based on the dependencies between device operations.

To address this challenge, Gupta and Sandhu introduced the Activity-Centric Access Control (ACAC) model in 2021 for smart and connected systems [5]. In this model, an activity refers to a long-lived operation performed by an IoT device. The ACAC model is designed to be object-agnostic, allowing flexibility in adding or removing devices as the system scales. An example activity is spraying water which is typically performed by a water sprinkler. Building upon this foundation, Mawla et al. later proposed key components of the ACAC model and incrementally enhanced it by introducing additional features, laying the groundwork for a more comprehensive ACAC model [6].

2.1 Activity-Centric Access Control Model

In smart and connected IoT-based systems, activities refer to long-running device operations, such as spraying water, plowing fields, and sowing seeds [7]. To enable real-time monitoring and control of these ongoing operations, the Activity-Centric Access Control (ACAC) model was proposed [5]–[7]. While basic authorization determines initial access requested by an entity, factors like environmental conditions, required one-time operations, and inter-activity dependencies are crucial for ensuring security and safety of the devices' activities. To address this, Mawla et al. introduced key decision components to regulate access throughout an activity's life cycle [6], [7].

- **Authorization (A):** The requesting source must go through an authorization mechanism to ensure that an unauthorized entity cannot access an activity.
- **Obligations (B):** Obligations are referred to as some required one-time operations either performed by the same requesting source or by any other entities of the system.
- **Conditions (C):** Conditions are presented for the environmental attributes that need to have certain values to allow access to the activity.

- **Dependencies (D):** Dependencies between activities are created due to the relationships such a sequential or concurrent execution, conflict between activities. Some dependent activities are required to be in certain states to allow the initiation or continuity of an activity.

An activity state belongs to one of the states from the set $\{inactive, dormant, aborted, running, hold, revoked, finished\}$ at each phase of its life cycle [7]. Initially, an activity is in the $inactive$ state. When a request for execution is received, it transitions to $dormant$ state. If the request is approved based on the decision parameters (A, B, C, D), and execution begins, the activity moves to the $running$ state. While running, an activity may transition to $revoked$, $finished$, or $hold$, depending on ongoing evaluations. Revocation occurs if the required decision parameters are no longer met, completion is determined when the activity fulfills its purpose, and an activity may be put on hold if an urgent request requires its temporary suspension. The activity decision components are evaluated at different stages: pre-decision parameters are checked before an activity starts, ongoing-decision parameters determine whether execution should continue or be revoked, and post-dependent activities are assessed to ensure they reach their required states after execution [7], [8].

3. $\mathrm{ACAC_{ABCD}}$ MODEL FOR ACTIVITY CONTROL

This section discusses the policy models governing activity control decision parameters: Authorization (A), Obligations (B), and Conditions (C). The essential $\mathrm{ACAC_D}$ model for handling dependencies (D) between activities has been discussed in [7] by Mawla and the co-authors. Our policy models for parameters A, B and C adapt the Attribute-Based Access Control (ABAC) [4] framework, a widely used access control model, to enforce policies for these parameters. We formally define the components and policies associated with A, B, and C, ensuring they accurately represent access requests and policy evaluations within the ACAC model. The models that represent access decisions based on parameters A, B, and C are denoted as $\mathrm{ACAC_A}$, $\mathrm{ACAC_B}$, and $\mathrm{ACAC_C}$, respectively.

3.1 $\mathrm{ACAC_A}$ - Authorizations Model

The authorization of requesting entities for a particular access has been a fundamental topic since the inception of access control mechanisms as part of security approaches. Unlike traditional access control models, which primarily focus on granting access to specified objects, ACAC is designed as an object-agnostic model, providing the flexibility to dynamically select the most suitable object based on requirements and availability. This ensures scalability by facilitating the addition or removal of devices in large ecosystems. The Zero Trust security architecture [21] follows the principle of "never trust, always verify", assuming that threats exist both inside and outside the network. Consequently, every request whether from internal or external users must undergo authorization, and continuous monitoring. To support this security paradigm, the two-step authorization process defined in Table I ensures granular access control, protecting both activities and corresponding devices in the system.

3.1.1 Model Components

The finite sets S, O, OP, and ACT represent sources, objects[1], operations, and activities in the system, respectively. Objects serve as formal representations of devices that execute activities. A source S requests access to an activity, typically to initiate or execute an activity ACT. The system then identifies the appropriate object and corresponding operation to execute the requested activity, as discussed in Mawla et al. [7]. The $\mathrm{ACAC_A}$ model incorporates policies to evaluate the authorization of a source (S) for an activity from the set ACT, as well as the authorization to perform an operation (OP) on an object (O) selected by the system. As illustrated in Table I, each entity in the system has associated attributes, where for each $att \in ATT$, a set of discrete values (defined as Range(att)) is assigned. Whether an attribute att is atomic-valued or set-valued is determined by the function $attType$.

TABLE I
$\mathrm{ACAC_A}$ MODEL DEFINITIONS FOR SOURCE AUTHORIZATION IN ACAC

Basic Sets and Functions:

- S, O, OP, ACT: Finite sets representing sources, objects, operations, and activities in the system, respectively.
- ATT: A finite set of attributes associated with entities and system-wide environmental attributes.
- Range(att): Defines a finite set of atomic values for each attribute $att \in ATT$.
- attType: A function that classifies each attribute as either set-valued or atomic-valued, formally defined as attType: $ATT = \{\text{set}, \text{atomic}\}$.
- POL: A set of authorization policies, where each policy consists of one or more rules for granting or denying activity access to a requesting source.
- Entities in S, O are mapped to attribute values for each attribute $att \in ATT$. Mathematically:

$$att : S \cup O \longrightarrow \begin{cases} 2^{\text{Range}(att)} & \text{if } attType(att) = \text{set} \\ \text{Range}(att) \cup \{\perp\} & \text{if } attType(att) = \text{atomic} \end{cases}$$

S-ACT Authorization Function: For each activity $act \in ACT$, the authorization function $Auth_{S-ACT}(s : S, act : ACT)$ is defined using the propositional logic described below.
S-O Authorization Function: For each object $o \in O$, the authorization function $Auth_{S-O}(s : S, o : O, op : OP)$ is defined using the propositional logic described below.
Authorization functions, whether S-ACT or S-O, are specified using propositional logic, following this grammar:

- α ::= expr | expr $\wedge$ expr | $\forall x \in$ set.α | atomic $\in$ set
- expr ::= atomic atomicAttValCompare atomic | set setAttValCompare set
- atomicAttValCompare ::= $\geq$ | $\leq$ | $=$
- setAttValCompare ::= $\subseteq$ | $\cap$ | $\cup$ | $\neq$ | $\emptyset$
- **Definition of atomic (two cases):**
  - S-ACT access: atomic ::= $att(s)$ | value; where $att \in ATT$, $s \in S$, $attType(att) =$ atomic
  - S-O access: atomic ::= $att(i)$ | value; where $att \in ATT$, $i \in S \cup O$, $attType(att) =$ atomic
- **Definition of set (two cases):**
  - S-ACT: set ::= $att(s)$ | setValue; $att \in ATT$, $s \in S$, $attType(att) =$ set
  - S-O: set ::= $att(i)$ | setValue; $att \in ATT$, $i \in S \cup O$, $attType(att) =$ set

Authorization Decision
A source $s \in S$ is permitted to perform an operation $op \in OP$ on an object $o \in O$ to access an activity $act \in ACT$, stated as $Authorization_{S-ACT}(s : S, act : ACT, o : O, op : OP)$, if the required policies necessary to allow the operation are evaluated to make the final decision on granting authorization to the source s. These multi-layer policies must be evaluated for $Auth_{S-ACT}(s : S, act : ACT)$ and $Auth_{S-O}(s : S, o : O, op : OP)$. Formally,

$$Authorization_{S-ACT}(s : S, act : ACT, o : O, op) \longrightarrow \{True, False\}$$

defined as $Authorization_{S-ACT}(s : S, act : ACT, o : O, op) \equiv Auth_{S-ACT}(s : S, act : ACT) \wedge Auth_{S-O}(s : S, o : O, op : OP)$

3.1.2 Model Definitions

The set of policies POL is specified using propositional logic based on the Backus-Naur Form (BNF) grammar. In

[1]Since, an activity is typically performed by an IoT device in smart ecosystems, we treat the terms object and device as equivalent in activity-centric access control.

the $\mathrm{ACAC_A}$ model, we employ a two-step authorization mechanism to evaluate access requests initiated by a source; S-ACT Authorization Function evaluates whether a source is authorized to access a given activity. S-O Authorization Function determines whether a source is authorized to perform an operation *op* on an object *o* to access an activity act. To handle atomic values, we apply the comparison operators greater than or equal to ($\geq$), less than or equal to ($\leq$), or equal to ($=$) in order to compare actual values against system-defined threshold values in the policies. For set-valued attributes (e.g., roles), entity values must comply with system-accepted values, enforced by operators such as subset inclusion ($\subseteq$), intersection ($\cap$), union ($\cup$), inequality ($\neq$), and empty set ($\emptyset$).

In the S-ACT authorization function, both atomic and set-valued attributes of sources (S) are evaluated according to policy rules. On the other hand, in the S-O authorization function, atomic and set-valued attributes of both sources (S) and objects (O) are evaluated. Authorization functions $S{-}ACT$ and $S{-}O$ are formally represented as $Auth_{S-ACT}(s : S,\ act : ACT)$ and $Auth_{S-O}(s : S,\ o : O,\ op : OP)$ respectively. Each function returns *True* or *False*, depending on policy evaluations. The final authorization decision function determines whether a source S is authorized to perform an operation OP on an object O to access an activity ACT and represented by $Authorization_{S-ACT}(s : S,\ act : ACT,\ o : O,\ op : OP)$. This authorization decision evaluates to *True* or *False* based on the results of $Auth_{S-ACT}(s : S,\ act : ACT) \wedge Auth_{S-O}(s : S,\ o : O,\ op : OP)$.

3.2 $\mathrm{ACAC_B}$ - Obligations Model

$\mathrm{ACAC_B}$ model introduces the tracking mechanism of the mandatory one-time operations where each operation is supposed to be performed by a subject (may be same requesting source or different for a requested activity) on a pre-defined object. These requirements are referred to as the obligations. The $\mathrm{ACAC_{preB}}$ sub-model verifies whether pre-obligations are fulfilled before allowing the activity to start, while the $\mathrm{ACAC_{onB}}$ sub-model ensures that ongoing obligations are met to maintain the continuity of the running activity.

3.2.1 Model Components

The sets ACT, OBS, OBO, and $OBOP$ represent finite sets of activities, obligation subjects, obligation objects, and obligation operations, respectively. The set OB denotes a finite collection of obligations, where each obligation is associated with an obligation subject (OBS), an obligation object (OBO), and an obligation operation ($OBOP$). Since obligations are enforced at two different stages of a requested activity, the sets of pre-obligations and ongoing obligations may differ. To obtain obligations at two different stages, we define the functions $getPreOB$ for pre-obligations and $getOnOB$ for ongoing obligations, respectively. The functions return an empty set if there are no obligations regarding the requested activity. Additionally, the $obFulfilled$ function determines whether a given obligation has been fulfilled or not.

TABLE II
$\mathrm{ACAC_B}$ MODEL DEFINITIONS FOR CHECKING OBLIGATION FULFILLMENT IN ACAC

Basic Sets and Functions:

- $ACT, OBS, OBO, OBOP$: Finite sets representing activities, obligation subjects, obligation objects, and obligation operations in the system, respectively.
- OB: A finite set of obligations, defined as a subset of $OBS \times OBO \times OBOP$, formally, $OB \subseteq OBS \times OBO \times OBOP$.
- $getPreOB$: $ACT \longrightarrow 2^{OB}$, mapping each activity to a set of pre-obligations.
- $getOnOB$: $ACT \longrightarrow 2^{OB}$, mapping each activity to a set of ongoing obligations.
- $obFulfilled$: $OB \longrightarrow \{True, False\}$, a function that maps an obligation $ob \in OB$ to *True* or *False*, indicating whether the obligation is fulfilled or not.

$\mathrm{ACAC_{preB}}$ Model Definition

Whether the pre-obligations for a requested activity $act \in ACT$ are fulfilled or not is denoted by the function $preB(act) \longrightarrow \{True, False\}$. The function $preB(act)$ is formally defined as $\bigwedge_{(ob \in getPreOB(act))} obFulfilled(ob)$, where $obFulfilled$ represents system-provided information ($True$ or $False$) regarding the fulfillment of an obligation $ob \in OB$.

$\mathrm{ACAC_{onB}}$ Model Definition

Whether the ongoing obligations for a requested activity $act \in ACT$ are fulfilled or not is denoted by the function $onB(act) \longrightarrow \{True, False\}$. The function $onB(act)$ is formally defined as $\bigwedge_{(ob \in getOnOB(act))} obFulfilled(ob)$, where $obFulfilled$ represents system-provided information ($True$ or $False$) regarding the fulfillment of an obligation $ob \in OB$.

3.2.2 Model Definitions

$\mathrm{ACAC_{preB}}$ is evaluated to ensure that pre-obligations are fulfilled before an activity starts. $\mathrm{ACAC_{onB}}$ verifies that ongoing obligations are fulfilled during the execution of an activity. The $\mathrm{ACAC_{preB}}$ model determines pre-obligation fulfillment using the functional predicate $preB(act)$, which evaluates the function $obFulfilled$ for all obligations retrieved using $getPreOB(act)$ for a given requested activity act. The function $preB(act)$ returns either True or False based on the evaluation. Similarly, the $\mathrm{ACAC_{onB}}$ model determines ongoing obligation fulfillment using the functional predicate $onB(act)$, which evaluates ongoing obligations using $obFulfilled$ for all obligations retrieved via $getOnOB(act)$ for a given activity act.

3.3 $\mathrm{ACAC_C}$ - Conditions Model

As discussed in previous sections, conditions are environmental restrictions that must be met either before or during the execution of a requested activity. In the ACAC model, conditions are defined using environmental attributes, which must match predefined values. These values may be evaluated using various rules, such as checking whether a value falls within a range, equals to a predefined value, or is greater than or equal to a threshold value.

3.3.1 Model Components

The finite set of activities (ACT) remains consistent across all models in ACAC. The set *COND* represents a finite set of environmental conditions, where each condition in *COND* is associated with an environmental attribute. To evaluate conditions before and during an activity's execution, functions $getPreCond$ and $getOnCond$ map an activity to one or more conditions from the condition set. Additionally, Range(att) defines a finite set of atomic values for each $att \in ATT$.

TABLE III
$\mathrm{ACAC_C}$ MODEL DEFINITIONS FOR CONDITION EVALUATIONS IN ACAC

Basic Sets and Functions:

- ATT: A finite set of attributes associated with entities and system-wide environmental attributes.
- *COND*: A finite set of environmental conditions, where each condition is defined using an attribute from ATT.
- $getPreCond$: $ACT \longrightarrow 2^{COND}$, maps each activity to a set of pre-conditions.
- $getOnCond$: $ACT \longrightarrow 2^{COND}$, maps each activity to a set of ongoing conditions.
- Range(att): Defines a finite set of atomic values for each attribute $att \in ATT$.
- attType: ATT = {set, atomic}, a function classifying each attribute as either set-valued or atomic-valued.
- Attributes associated with a *COND* are mapped to attribute values for every attribute $att \in ATT$. Mathematically,

$$att : COND \longrightarrow \begin{cases} 2^{\text{Range}(att)} & \text{if } attType(att) = \text{set} \\ \text{Range}(att) \cup \{\perp\} & \text{if } attType(att) = \text{atomic} \end{cases}$$

Condition Fulfillment Function:

For each condition $cond \in COND$, the function $condSatisfied$ determines whether the condition is fulfilled, based on the propositional logic defined below.

The condition fulfillment function $condSatisfied$ is expressed using propositional logic, following the grammar below:

- α ::= expr | expr $\wedge$ expr | $\forall x \in$ set.α| atomic $\in$ set
- expr ::= atomic atomicAttValCompare atomic | set setAttValCompare set
- atomicAttValCompare ::=$\geq$ | $\leq$ | =
- setAttValCompare ::= $\subseteq$ | $\cap$ | $\neq$ |$\emptyset$
- atomic ::= *att*(i) | value
  where for each $att \in ATT$, $i \in COND$, and $attType(att) =$ atomic
- set ::= *att*(i) | setValue
  where for each $att \in ATT$, $i \in COND$, and $attType(att) =$ set

$\mathrm{ACAC_{preC}}$ Model Definition

Whether the pre-conditions for a requested activity $act \in ACT$ are satisfied or not is denoted as $preC(act) \longrightarrow \{True, False\}$. The function $preC(act)$ is formally defined as $\bigwedge_{(cond \in getPreCond(act))} condSatisfied(cond)$, where $condSatisfied$ is evaluated for each condition $cond \in getPreCond(act)$.

$\mathrm{ACAC_{onC}}$ Model Definition

Whether the ongoing conditions for a requested activity $act \in ACT$ are satisfied or not is denoted as $onC(act) \longrightarrow \{True, False\}$. The function $onC(act)$ is formally defined as $\bigwedge_{(cond \in getOnCond(act))} condSatisfied(cond)$, where $condSatisfied$ is evaluated for each condition $cond \in getOnCond(act)$.

In $\mathrm{ACAC_C}$ model, each att associated with a condition in *COND* is mapped to attribute values based on the attribute type.

3.3.2 Model Definitions

The $\mathrm{ACAC_C}$ model defines a framework for evaluating environmental conditions that must be satisfied before or during the execution of an activity in the ACAC system. The model introduces two sub-models: $\mathrm{ACAC_{preC}}$, which evaluates pre-conditions that must be met before an activity starts, and $\mathrm{ACAC_{onC}}$, which evaluates ongoing conditions that must be fulfilled to maintain the execution of a running activity. The finite set *COND* represents environmental conditions, each associated with an attribute from ATT, with conditions mapped to activities using the functions $getPreCond$ and $getOnCond$.

The fulfillment of a condition is determined using $condSatisfied$, a propositional logic function that evaluates conditions based on policies. For atomic attributes, conditions are enforced using relational comparisons such as greater than or equal to ($\geq$), less than or equal to ($\leq$) and equal to a predefined value, while set-valued attributes use operators such as subset inclusion ($\subseteq$), intersection ($\cap$), and inequality ($\neq$). The model formally defines the evaluation of pre-conditions and ongoing conditions using $\bigwedge_{(cond \in getPreCond(act))} condSatisfied(cond)$ and $\bigwedge_{(cond \in getOnCond(act))} condSatisfied(cond)$, and presented by the functional predicates $preC$ and onC respectively. These functional predicates return $True$ or $False$, determining whether an activity can start or continue based on system-defined environmental conditions.

4. $\mathrm{ACD_{ACAC_{ABCD}}}$: ACTIVITY CONTROL DECISION ALGORITHM FOR CONSOLIDATED $\mathrm{ACAC_{ABCD}}$ MODEL

An access control model requires a well-defined algorithm to systematically evaluate multiple parameters, ensuring consistent decision-making and enforcement of security policies. In this section, we propose the Activity Control Decision Algorithm (ACD) for implementing the ACAC model considering all of the proposed decision components Authorizations (A), Obligations (B), Conditions (C) and Dependencies (D) between activities. We refer to the algorithm as $\mathrm{ACD_{ACAC_{ABCD}}}$. The algorithm is implemented to develop the policy engine, where policy evaluations are conducted using the models defined for all parameters in ACAC.

As depicted in Figure 1, a source s initially requests an activity act. As the first step of the two step authorization approach, the authorization of the source s to initiate activity act is evaluated using the propositional logic defined for the function $Auth_{S-ACT}(s : S, act : ACT)$ in Table I. The policy engine looks for a suitable and available object o for running the activity act. If there is no available object found, the requested activity act is aborted. Otherwise, the policy engine looks for an operation op to perform on the object o in order to run act. At this stage, the $\mathrm{ACD_{ACAC_{ABCD}}}$ algorithm engine evaluates the policies defined in the $\mathrm{ACAC_A}$ model to determine whether the source s is authorized to perform the operation op on the object o. If the source s is authorized, meaning it is permitted to perform op on o to execute act, the ACD engine retrieves the pre-obligations and pre-conditions from the system-defined policies. These parameters are then evaluated using the policies defined in the $\mathrm{ACAC_{preB}}$ and $\mathrm{ACAC_{preC}}$ models, as outlined in Tables II and III.

If any pre-obligation or pre-condition is not satisfied, the activity act will be $aborted$. Otherwise, the engine identifies the pre-dependent activities along with their desired states and verifies whether all pre-dependent activities are in their required states. If all dependent activities are in their desired states, the activity is allowed to transition to $running$ state. However, if some dependent activities are not in their desired states, they must be updated before act can start execution. As discussed in [7], dependent activities may have their own dependencies when transitioning from one state to another (e.g., $inactive$ to $running$), making state updates non-trivial due to the need for verifying the dependencies of dependencies. To address this, the ACD engine recursively applies the same algorithm to dependent activities. Once all dependent activities successfully transition to their desired states, requested activity act is permitted to run. However, if any dependent activity is immutable and remains in a state other than the desired state, the requested activity will be aborted. If all dependent activities reach their desired states, requested activity starts execution. Finally, before allowing the requested activity act to run, the system must ensure that all relevant constraints

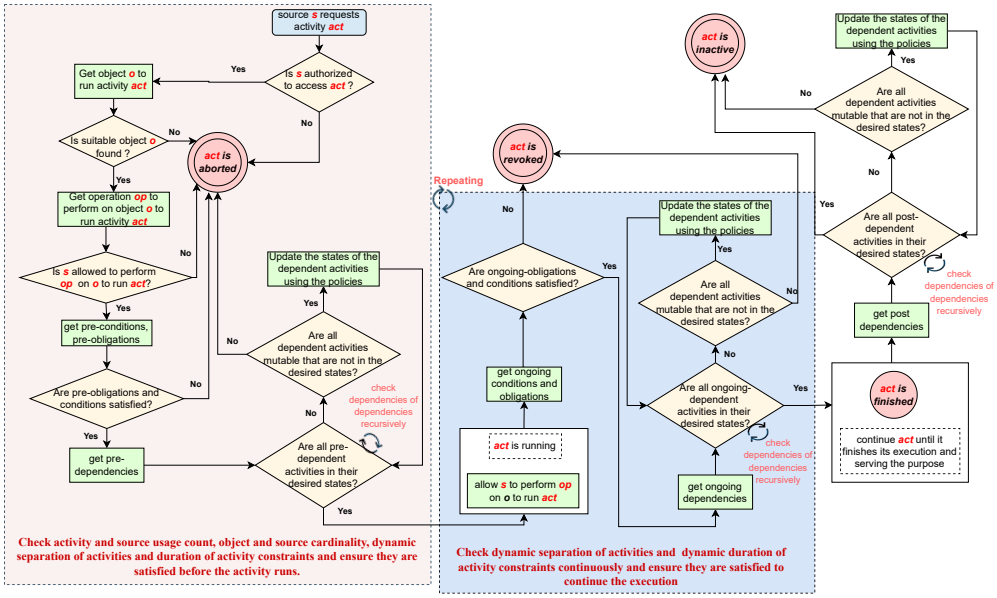

Figure 1. Flow Chart for Activity Control Decision ($\mathrm{ACD}_{\mathrm{ACAC}_{\mathrm{ABCD}}}$) Algorithm

are satisfied, as outlined in [9] by Mawla and Gupta. These include source and activity usage count, dynamic separation of activities, and source and object cardinality. If any of these constraints exist for act, they must be evaluated and satisfied before execution can proceed.

When the activity is allowed to run, it enters the ongoing phase. The ACD engine retrieves the ongoing obligations and conditions, which are evaluated using the $\mathrm{ACAC}_{\mathrm{onB}}$ and $\mathrm{ACAC}_{\mathrm{onC}}$ models, as defined in Tables II and III. If these obligations and conditions are satisfied, the engine then retrieves and checks ongoing dependent activities. The dependent activity states verification and updates follow a similar process to the pre-execution phase, with policy evaluations provided by the $\mathrm{ACAC}_{\mathrm{onD}}$ model for ongoing dependent activities. [7]. As illustrated in Figure 1, the middle blue section represents a repeating process during the execution of the requested activity. This repetition occurs because any violation of policies associated with A (Authorization), B (Obligations), C (Conditions), or D (Dependencies) results in the activity transitioning to a revoked state. To ensure continuity, these ongoing evaluations are performed throughout the duration of the activity. Additionally, constraints such as dynamic separation of activities and dynamic duration enforcement are continuously checked during execution to maintain the policy.

Once the execution of the requested activity is finished, the engine ACD retrieves the post-dependent activities and update the states of the post dependent activity states if they are not in their desired states. During the update procedure, each update follows the similar algorithm as provided for the requested activity. After the post-update procedure the activity again goes to its *inactive* state.

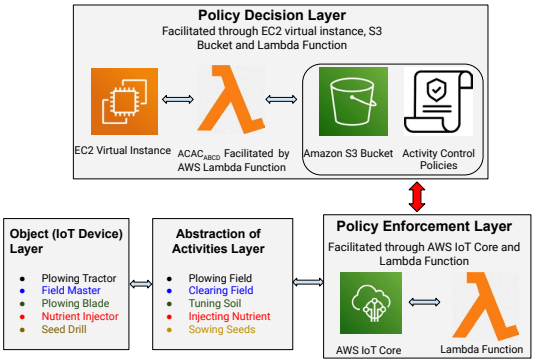

Figure 2. Implementation Architecture

5. Proof of Concept Implementation

This section presents a proof-of-concept implementation of the proposed $\mathrm{ACAC}_{\mathrm{ABCD}}$ model using Amazon Web Services (AWS) cloud services. The implementation showcases how the security policies defined within the ACAC model enhance security and safety in complex and connected systems. Additionally, we evaluate the performance of the ACAC model by comparing it with a related Attribute-Based Access Control (ABAC) model, using a simulated smart farming use case to evaluate their effectiveness in managing security within interconnected IoT environments.

5.1 System Architecture

The complete architecture of the prototype implementation is shown in Figure 2. The demonstrated architecture is facilitated by the Activity Control Decision Algorithm ($\mathrm{ACD}_{\mathrm{ACAC}_{\mathrm{ABCD}}}$), detailed in Section 3. We utilize the commercially available

ACT = ACT_R = ACT_D = {$fieldPlowing$, $clearingField$, $treeRemoval$, $collectingDebris$, $surfaceLeveling$, $tuningSoil$, $adjustingBladeDepth$, $sowingSeeds$, $selectingSeeds$, $pesticideSpray$, $waterSpray$, $dripIrrigation$, $pumpingWater$, $pipelinePressureTesting$, $soilMoistureRegulation$, $irrigationCoverageAdjustment$, $injectingNutrient$, $cleaningPipeline$, $mixingNutrient$, $moistureSensorCalibration$}, ***ST*** = ST_{CR} = ST_{DR} = {$inactive$, $dormant$, $aborted$, $running$, $hold$, $revoked$, $finished$}
S = {$Daniel$, $David$, $Ella$, $Emily$, $Emma$, $Ethan$, Eva, $George$, $Grace$, $Hannah$, $Henry$, $Isabella$, Ivy, $Jack$, $Jacob$, $James$, $John$, Jon, $Kevin$, $Layla$, $Liam$, $Lily$, $Lucas$}
O = {$Emitter$, $CentrifugalPump$, $pressureTester$, $moistureRegulator$, $valveController$, $nutrientInjector$, $pipelineCleaner$, $calibrationTool$, $mixingMachine$, $spreaderTractor$, $plowingTractor$, $soilAerator$, $compactionRoller$, $spreader$, $injector$, $autoWeedX$, $fieldMaster$, $treeHarvester$, $debriSweep$, $terraLevelPro$, $spreadTune$, $plowingBlade$, $depthMaster$, $seedDrill$, $seedMaster$, $pestSprayDrone$, $sprinkler$, $loadMaster$}, ***OP*** = {$turnOn$, $turnOff$}

OBS = {Jon, $Emily$, $Mike$, $Sarah$, $David$, $Anna$, $Sophia$, $James$, $Ethan$, $Grace$}, ***OBO*** = {$waterPump$, $irrigationValve$, $waterSourceValve$, $plowBlades$, $plowingMachine$}, ***OBOP*** = {$turnOn$, $adjust$, $open$, $initialize$, $close$, $load$, $adjustAngle$, $fill$, $start$, $activate$, $release$, $calibrate$, add, $setDepth$, $turnOff$, $empty$, $unload$, set, $apply$ }

COND = {$soilType == loamy$, $(plowingDepth \geq 15) \wedge (plowingDepth \leq 25)$, $(plowingDepth \geq 15) \wedge (plowingDepth \leq 30)$, $(PH \geq 6) \wedge (PH \leq 7)$, $soilMoisture < 50$, $pipelinePressure > 1.5$}

$request = (Ethan, fieldPlowing)$

$Auth_{Ethan-fieldPlowing}(Ethan, fieldPlowing) \equiv ((distance(Ethan) \geq 1) \wedge (distance(Ethan) \leq 2)) \wedge (group(Ethan) == field12) \wedge (role(Ethan) == plowing-technician)$
$getObject(fieldPlowing) = plowingTractor$
$getOperation(fieldPlowing, plowingTractor) = turnOn$

$Auth_{Ethan-plowingTractor}(Ethan, plowingTractor, turnOn) \equiv (type(plowingTractor) == sensitive \wedge role(Ethan) \in \{farm-manager, irrigation-specialist, water-sprayer-technician, fertilizer-specialist, weed-removal-specialist, clearing-specialist, tree-removal-expert, surface-leveling-expert, spreader-technician, soil-tuning-expert, pesticide-expert\}) \vee (type(plowingTractor) == regular \wedge role(Ethan) \in \{farm-manager, irrigation-specialist, water-specialist, farm-worker, technician, engineer, irrigation-technician, field-supervisor, agronomist, maintenance-engineer, sensor-specialist, aeration-specialist, plowing-technician, compaction-expert, fertilizer-specialist, micronutrient-technician, weed-removal-specialist, clearing-specialist, tree-removal-expert, debris-collector, surface-leveling-expert, spreader-technician, soil-tuning-expert, blade-adjustment-technician, seed-sowing-specialist, seed-sorting-technician, pesticide-expert, water-sprayer-technician, raw-material-loader\})$

$Authorization_{Ethan-fieldPlowing}(Ethan, fieldPlowing, plowingTractor, turnOn) \equiv Auth_{Ethan-fieldPlowing}(Ethan, fieldPlowing) \wedge Auth_{Ethan-plowingTractor}(Ethan, plowingTractor, turnOn)$

$getPreOB(fieldPlowing) = \{(Ethan, plowBlades, setDepth)\}$
$getPreCond(fieldPlowing) = \{soilType == loamy\}$
$getDA(fieldPlowing, plowingTractor) = \{clearingField\}$

$getDesiredPreDASt(fieldPlowing, clearingField) = finished$

$preD(fieldPlowing, plowingTractor) \equiv getCurrentSt(clearingField) == finished$

$preUpdate(fieldPlowing, plowingTractor) \Rightarrow preD(fieldPlowing, plowingTractor) == False$

$has\_DSA(fieldPlowing, pesticideSpray) \Rightarrow condSatisfied((soilMoisture < 50))$

$actUsageCount(fieldPlowing) = 2$

$sourceUsageCount(fieldPlowing) = \{Jon : 2, Emily: 2\}$

$objectCardinality(fieldPlowing) = 5$

$constraintsSatisfied(fieldPlowing, pre) \equiv checkConstraints(fieldPlowing, pre)$

$allowed(Ethan, plowingTractor, turnOn, fieldPlowing) \equiv Authorization_{Ethan-fieldPlowing}(Ethan, fieldPlowing, plowingTractor, turnOn) \bigwedge obFulfilled((Ethan, plowBlades, setDepth)) \bigwedge condSatisfied((soilType, equal\_to, loamy)) \bigwedge preD(fieldPlowing, plowingTractor) \bigwedge constraintsSatisfied(fieldPlowing, pre)$

$getOnOB(fieldPlowing) = \{(Grace, plowingMachine, turnOn)\}$
$getOnCond(fieldPlowing) = \{(plowingDepth \geq 15) \wedge (plowingDepth \leq 25)\}$

$getDA(fieldPlowing, plowingTractor) = \{tuningSoil, injectingNutrient, sowingSeeds\}$ $getDesiredOnDASt(fieldPlowing, tuningSoil) = running$
$getDesiredOnDASt(fieldPlowing, injectingNutrient) = finished$

$onD(fieldPlowing, plowingTractor) \equiv getCurrentSt(tuningSoil) == running \bigwedge getCurrentSt(injectingNutrient) == finished$

$onUpdate(fieldPlowing, plowingTractor) \Rightarrow onD(fieldPlowing, plowingTractor) == False$

$has\_DSA(fieldPlowing, pesticideSpray) \Rightarrow condSatisfied((soilMoisture, less\_than, 50))$

$duration((fieldPlowing)) = 0.2$

$DDuration(fieldPlowing) = 0.1 \Rightarrow condSatisfied((soilMoisture < 50))$

$constraintsSatisfied(fieldPlowing, ongoing) \equiv checkConstraints(fieldPlowing, ongoing)$

$stopped(fieldPlowing, plowingTractor) \Rightarrow \neg obFulfilled(Grace, plowingMachine, turnOn) \vee \neg condSatisfied((plowingDepth, in\_between, [15, 25])) \vee \neg onD(fieldPlowing, plowingTractor) \vee \neg constraintsSatisfied(fieldPlowing, ongoing)$

$getDA(fieldPlowing, plowingTractor) = \{sowingSeeds\}$
$getDesiredPostDASt(fieldPlowing, sowingSeeds) = running$
$postD(fieldPlowing, plowingTractor) = getCurrentSt(injectingNutrient) == running$
$postUpdate(fieldPlowing, plowingTractor) \Rightarrow postD(fieldPlowing, plowingTractor) == False$

Figure 3. Smart Farming Use Case Configuration including Policies for Authorizations, Obligations, Conditions and Dependencies

AWS IoT Services to facilitate the storage and enforcement of the policies. AWS IoT things are created in AWS IoT Core[2] service where we defined two attributes (availability and status) for each thing. Each thing is representing an IoT device. Since the primary goal of our model is to control activities, which serve as abstractions for devices' long-continuous operations, we do not explicitly address device statuses within the model. However, in the implementation, the availability and status of corresponding devices are updated in sync with activity state transitions as defined in the use case. This approach is intentionally adopted to simulate device behavior and illustrate the relationship between devices and their respective activities within the system. In Figure 2, the corresponding devices which are suitable to perform the activities (shown in Abstraction of Activities Layer), are shown in the Object (IoT Device) Layer. We represented each activity and its corresponding device using the same color for clarity. However, in our use cases, additional devices are also utilized beyond those explicitly shown.

We utilize the AWS S3 Bucket service[3] to store all policies related to authorizations, obligations, conditions, and dependencies between activities. Additionally, real-time and original data, including source and condition attribute values, obligation fulfillment status, current states of all activities, object information, and availability, are continuously updated in JSON files configured to manage this information. The JSON files reside in the S3 Bucket.

An EC2 virtual instance[4] was set up to serve as an intermediary between the requesting source and the Lambda function. Each request consists of a source and a requested activity. A Python script was developed to send these requests from the EC2 instance to the Lambda function. The Lambda function[5] is responsible for facilitating the implementation of the $\mathrm{ACD}_{\mathrm{ACAC}_{\mathrm{ABCD}}}$ model. The Lambda function was implemented as a policy engine using Python and the Boto3 library to interface with the EC2 application and process JSON files stored in an S3 bucket. IAM roles were configured for the Lambda function, with appropriate policies attached to grant the necessary permissions for S3 access. Similarly, the EC2 instance was configured with permissions to interact with the Lambda function. Since activities serve as abstractions of device operations, any state changes in activities that are implemented logically through policies in the Lambda function are reflected in the corresponding devices (objects). For instance, when an activity is initiated on a device through an operation performed by a source, the device status transitions from "off" to "on", and its availability updates from "True" to "False" with the corresponding update made in the JSON configured for objects. To facilitate this interaction, the Lambda function is granted the necessary permissions to access IoT Things within AWS IoT Core Service.

[2]https://aws.amazon.com/iot-core/
[3]https://aws.amazon.com/s3/
[4]https://aws.amazon.com/ec2/
[5]https://aws.amazon.com/lambda/

5.2 Use case Scenarios

This section introduces a smart-farming use case for a requested activity $fieldPlowing$ shown in Figure 3. In this use-case configuration, we first listed the sets (black-coloured texts) of all distinct activities (ACT), states of activities (ST), sources (S), objects (O), obligation subjects (OBS), obligation objects (OBO), obligation operations ($OBOP$) and conditions (*COND*) that are present in the farming scope. For each requested activity, the requested and dependent activities are obtained from the set ACT, and suitable objects for the activities are retrieved from the set O. At any given point of time, each activities belongs to one of the states listed in set ST. Any requesting source must be included in the set S. Each obligation corresponding to an activity is defined using a subject, object and operation obtained from the sets OBS, OBO, and $OBOP$, respectively. The conditions are listed as comma-separated in set *COND*, where each condition has an environmental attribute and pre-defined rule for the attribute value. For example, a condition $PH \leq 7$ corresponds to the environmental attribute PH and the rule for PH is that it should be less than or equal to 7.

In our proof-of-concept implementation, we arrange to send different numbers of concurrent requests (at most 60). To configure the policies for all 60 activities, first, we generate thoughtful and practically useful policies for 20 distinct activities. Later, we used 40 more dummy activities and policies utilizing the same policy structure. Some of the requested activities include common dependent activities, while some of the dependent activities are, at some point, used as requested activities. In the use-case illustrated in Figure 3, we show the policies and evaluation of the policies for a requested activity $fieldPlowing$ which is requested by the source $Ethan$.

As illustrated in the previous subsection, in the implementation architecture, the AWS Lambda function is configured as the policy engine, which sequentially (following the ACD algorithm described in the Section 4) executes the models ($\mathrm{ACAC}_{\mathrm{A}}$, $\mathrm{ACAC}_{\mathrm{B}}$, $\mathrm{ACAC}_{\mathrm{C}}$ as detailed in Section 3, and $\mathrm{ACAC}_{\mathrm{D}}$ as described in [7]) to evaluate policies defined for authorizations (A), obligations (B), conditions (C), and dependencies (D) between activities. Additionally, the policy engine assesses the constraints specified in [9] during both the pre- and ongoing phases of a requested activity. Notably, to fulfill a requested activity, its dependent activities must undergo their respective life-cycle state transitions to reach the desired state. Policies are configured for all activities and are enforced for any state change associated with a requested activity from the set of 60 activities.

As configured in the use-case scenario, for the activity $fieldPlowing$, which is requested by the source $Ethan$, the policies for all decision parameters (A, B, C, D) need to be evaluated. According to the authorization model $\mathrm{ACAC}_{\mathrm{A}}$, the two-step source-authorization policies must be evaluated for the requesting source $Ethan$. As we see in the use-case figure, the authorization of $Ethan$ to access the requested activity $fieldplowing$ is evaluated using the function

$Auth_{Ethan-fieldPlowing}(Ethan, fieldPlowing)$ and returns $True$ or $False$ depending on the policy evaluation using the rule saying that $Ethan$ is authorized to access $fieldPlowing$ if the distance of $Ethan$ and the field is in between 1 and 2 meters, the group $Ethan$ belongs to is $field12$ and the role of $Ethan$ is a $plowing - technician$. The rule is formally stated as $Auth_{Ethan-fieldPlowing}(Ethan, fieldPlowing) \equiv ((distance(Ethan) \geq 1) \wedge (distance(Ethan) \leq 2)) \wedge (group(Ethan) == field12) \wedge (role(Ethan) == plowing - technician)$. As described in model ACAC_D in [7] by Mawla et al., the object (which is the formal term for the executing device) is retrieved from the system based on the suitability and availability. The object retrieval is done after evaluating the requesting source $Ethan$'s authorization to access $fieldPlowing$. The function $getObject(plowingField)$ is called in the policy engine to retrieve the available object which is in this case $plowingTractor$. Later, the required operation to execute the activity $fieldPlowing$ on $plowingTractor$ is obtained using the function $getOperation(fieldPlowing, plowingTractor)$. The operation involved here is $turnOn$.

Furthermore, the second step of the authorization process is evaluated to enhance the system's security. In this step, we assess whether $Ethan$ is authorized to perform the operation $turnOn$ on the device $plowingTractor$. The authorization rule is evaluated based on the type of the device. If $type(plowingTractor) = sensitive$, then $role(Ethan)$ must belong to a predefined set of roles, for example, $farm - manager$, $irrigation - specialist$, and more. However, if $type(plowingTractor) = regular$, then $role(Ethan)$ must belong to a broader set of roles. The function that evaluates this rule is represented as $Auth_{Ethan-plowingTractor}(Ethan, plowingTractor, turnOn)$ determining whether $Ethan$ meets the required role conditions to execute the operation. Finally, the authorization to allow $Ethan$ to perform an operation $turnOn$ on the object $plowingTractor$ to execute $fieldPlowing$ evaluates to $True$ if $Auth_{Ethan-fieldPlowing}(Ethan, fieldPlowing) \wedge Auth_{Ethan-plowingTractor}(Ethan, plowingTractor, turnOn)$ evaluates to $True$. The final authorization function is formally stated as $Authorization_{Ethan-fieldPlowing}(Ethan, fieldPlowing, plowingTractor, turnOn)$.

The set of pre-obligations for the requested activity $fieldPlowing$ is retrieved using the function $getPreOB(fieldPlowing)$, which returns a set of obligations. In this case, it includes a single obligation: $(Ethan, plowBlades, setDepth)$. If this obligation is fulfilled in the system, $fieldPlowing$ can be executed, provided that all other required parameters are satisfied. The preconditions for $fieldPlowing$ are retrieved using the function $getPreCond(fieldPlowing)$, which returns $(soilType == loamy)$. This indicates that the value of the condition attribute $soilType$ must be equal to $loamy$ for $fieldPlowing$ to proceed. To verify dependencies between activities and their states, we first retrieve the pre-dependent activities of $fieldPlowing$ using the function $getDA(fieldPlowing, plowingTractor)$, which returns the set $clearingField$. The function $getDesiredPreDASt(fieldPlowing, clearingField)$ then determines the desired state of the pre-dependent activity $clearingField$, which must be $finished$. The functional predicate $preD(fieldPlowing, plowingTractor)$ evaluates to $True$ if the current state of $clearingField$ matches the desired state, formally expressed as $getCurrentSt(clearingField) == finished$. If it returns $False$, the function $preUpdate(fieldPlowing, plowingTractor)$ is invoked to update the state of $clearingField$. The pre-update process follows the update methodology described in [7].

The state transitions of an activity are discussed in Section 2. To update the state of dependent activities when they are not in their desired states, we define the next state of each activity based on its current and desired states for implementation purposes. During the transition process, a dependent activity is treated like a requested activity, and policies are evaluated to determine the necessary transitions. If the policies prevent a dependent activity from reaching its desired state, the parent requested activity cannot transition into its required state. Consequently, the root requested activity will either not start (in the case of a pre-dependency) or be revoked (in the case of an ongoing dependency). The policies governing activity updates are defined separately for each activity and must be evaluated at any point in the activity's life cycle. However, after verifying the desired states of dependent activities, the necessary updates for different dependent activities are executed concurrently. This concurrent update mechanism is a key aspect of our implementation, effectively simulating real-time scenarios.

The constraints that must be checked before the requested activity $fieldPlowing$ starts and during its execution are highlighted in cyan-coloured text in Figure 3. During policy evaluation before the activity begins, constraints are assessed for dynamic separation of activities (DSA), activity usage count, source usage count, and object cardinality. The function $has\_DSA(fieldPlowing, pesticideSpray)$ returns $True$ (meaning $fieldPlowing$ and $pesticideSpray$ have dynamic separation of activity relation) if the condition on the right side of the implication holds. The activity usage count is set to 2, while the source usage count for the requested activity $fieldPlowing$ is also 2. Additionally, the object cardinality for $fieldPlowing$ is restricted to 5. These constraints are verified using the function $checkConstraints(fieldPlowing, pre)$, and based on the result ($True$ or $False$), the function $constraintsSatisfied(fieldPlowing, ongoing)$ determines whether the constraints are satisfied before execution, returning either $True$ or $False$.

Finally the function $allowed(Ethan, plowingTractor, turnOn, fieldPlowing)$ returns $True$ if $Authorization_{Ethan-fieldPlowing}(Ethan, fieldPlowing, plowingTractor, turnOn) \wedge obFulfilled((Ethan, plowBlades, setDepth)) \wedge condSatisfied((soilType$

$==$ $loamy$)) $\wedge$ $preD(fieldPlowing, plowingTractor)$ $\wedge$ $constraintsSatisfied(fieldPlowing, pre)$ returns $True$. The $allowed(Ethan, plowingTractor, turnOn, fieldPlowing)$ function stated by Mawla [7] only checks the dependencies whereas we get the complete evaluation of the function in this work. If $allowed(Ethan, plowingTractor, turnOn, fieldPlowing)$ returns $True$, the requested activity $fieldPlowing$ is permitted to be executed by the device $plowingTractor$. As mentioned the system architecture, the devices are configured as "Things" in AWS IoT Core and hold only two attributes "status" (either on or off) and "availability" (either $True$ or $False$). Once the function returns $True$, the status of the corresponding device (in this case, $plowingTractor$) transitions from its initial state off to on, reflecting the operation $turnOn$ performed by the source $Ethan$. Consequently, the current state of $fieldPlowing$ changes from $inactive$ (the default state for activities that have not yet been requested) to $running$.

If not denied, the current state of $fieldPlowing$ transitions to $running$. According to the ACAC model and the state transitions within a requested activity's life cycle, once the requested activity is allowed to run, the ongoing decision parameters must be periodically assessed to ensure its continuity. The ongoing policies are written in the use-case in purple-coloured text including cyan-coloured texts for constraints. As part of this evaluation, the ongoing obligations, conditions, and dependencies for the requested activity $fieldPlowing$ are retrieved using the functions $getOnOB(fieldPlowing)$, $getOnCond(fieldPlowing)$, and $getDA(fieldPlowing, plowingTractor)$, respectively. The evaluations of these parameters—ongoing obligations, conditions, dependencies, and constraints—are performed similarly to those in the activity's pre-execution phase.

Finally, we determine whether the continuity of $fieldPlowing$ should be stopped (meaning whether the activity should be revoked) by evaluating the function $stopped(fieldPlowing, plowingTractor)$. This function returns $True$ if the following condition holds: $\neg obFulfilled((Grace, plowingMachine, turnOn))$ $\vee$ $\neg condSatisfied((plowingDepth, in\_between, [15, 25]))$ $\vee$ $\neg onD(fieldPlowing, plowingTractor)$ $\vee$ $\neg constraintsSatisfied(fieldPlowing, ongoing)$ (meaning if any of the ongoing decision parameters is not satisfied). If the execution of the requested activity $fieldPlowing$ is successfully completed, the post-dependent activities are retrieved using $getDA(fieldPlowing, plowingTractor) = sowingSeeds$. The required updates are then processed, but only if necessary when $postD(fieldPlowing, plowingTractor)$ returns $False$ by invoking $postUpdate(fieldPlowing, plowingTractor)$. Upon successful completion of the full activity cycle, the AWS Lambda function returns $True$ to the EC2 instance as part of the response criteria.

TABLE IV
PERFORMANCE METRICS FOR EC2 TO LAMBDA INTERACTION. RTL DENOTES ROUND TRIP LATENCY, NDC AND NDU DENOTE NUMBER OF DEPENDENCIES CHECKED AND UPDATED RESPECTIVELY.

Number of Concurrent Requests	Avg. RTL (ms)	Lambda Time (ms)	NDC	NDU
1	7396.56	773.44	2	2
5	8023.07	1255.20	3	1
10	10436.72	2618.89	5	2
20	17411.67	2294.12	30	95
30	24525.66	2913.96	44	138
40	30752.28	2801.79	58	186
50	38470.65	2941.43	74	235
60	45081.93	2890.98	90	279

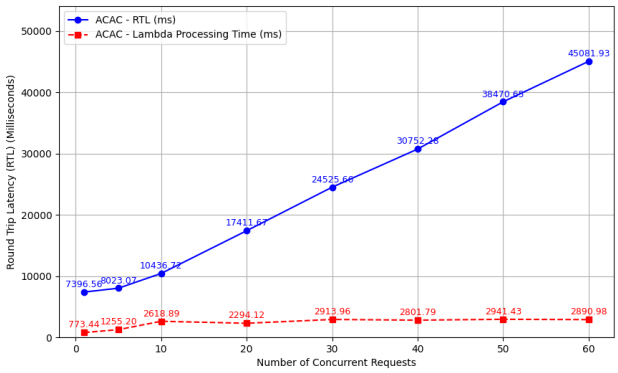

Figure 4. The round trip latency (RTL) and lambda processing time for $\text{ACAC}_{\text{ABCD}}$ model against number of concurrent Requests

5.3 Performance Evaluation

To evaluate the effectiveness of the proposed ACAC model, we conducted a comprehensive performance analysis and compared it against the ABAC model using the defined use case scenario. The implementation was executed as an AWS Lambda function in the cloud, enabling scalable and on-demand processing while requiring careful consideration of memory allocation and execution time for optimization. However, since the Lambda function operates in a cloud environment, its performance may be affected by factors such as internet connectivity, AWS resource availability, and potential network-related delays. These aspects should be considered when evaluating processing time, as fluctuations may result from cloud infrastructure conditions rather than inefficiencies in the model itself.

The $\text{ACD}_{\text{ACAC}_{\text{ABCD}}}$ algorithm engine was tested under varying numbers of concurrent requests, and key performance metrics were recorded. Our evaluation considered 1, 5, 10, 20, 30, 40, 50, and 60 concurrent requests, ensuring different number of pre-, ongoing-, and post-dependent activities. This setup allowed us to analyze the system's behavior under various dependency structures, particularly how dependency checks and updates affect processing time. The performance analysis focused on four primary metrics: (i) Average Round Trip Latency (RTL) between the EC2 application and the Lambda function, where the Lambda function serves as the

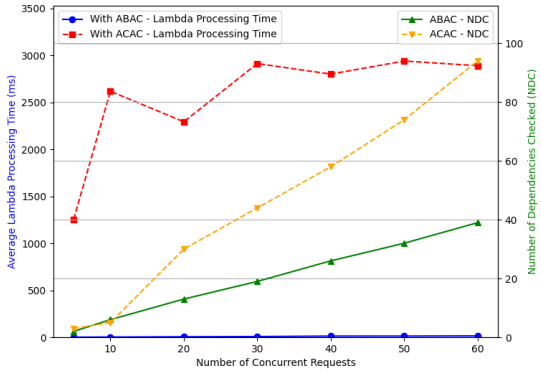

Figure 5. A comparison of the performance between ABAC and ACAC models in terms of Lambda processing time and the number of dependencies checked.

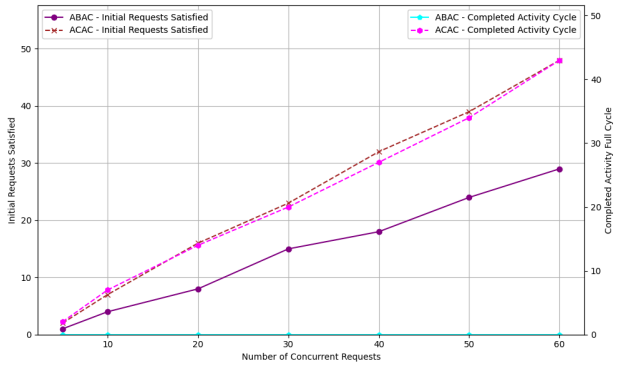

Figure 6. A comparison of the performance between ABAC and ACAC models in terms of the number of initial requests satisfied and the number of fully completed activity cycles.

policy engine facilitating $\mathrm{ACD}_{\mathrm{ACAC}_{\mathrm{ABCD}}}$ algorithm, (ii) Average Lambda processing time, (iii) Number of Dependencies Checked (NDC), and (iv) Number of Dependencies Updated (NDU). As summarized in Table IV, the results indicate that NDC and NDU increase significantly with a higher number of concurrent requests. This occurs because as more activities are requested, a greater number of dependencies need to be verified and updated. Additionally, the round-trip latency (RTL) increases with concurrent requests, though its growth pattern does not follow a strictly linear or exponential trend, as shown in Figure 4. The scalability of the ACAC model was a critical aspect of our evaluation. Our results demonstrate that the proposed ACAC policy engine maintains stable performance without significant degradation as the number of concurrent requests increases. This is a key advantage, indicating that ACAC efficiently scales with increased workloads. Moreover, the update process for activity states contributes to processing time variations, as it involves modifying the status of IoT devices. Our implementation used AWS IoT Core "Things" configured with two attributes: "Availability" and "Status" to reflect real-time state changes for activities. This integration highlights the practicality of ACAC, demonstrating that the model can be effectively implemented in real-world IoT environments. However, the total processing time will depend on the duration of requested and dependent activities, which was fixed at 200 milliseconds in our evaluation. A direct performance comparison between ABAC and ACAC models further reveals key differences in handling access control decisions. In ABAC, access requests are processed passively, where a subject requests access to an object, and access is granted based on attribute evaluations. However, ABAC is not object-agnostic, making it less suitable for large-scale IoT environments where devices are frequently added or removed. Additionally, ABAC does not support ongoing access control, meaning it cannot reevaluate decisions during execution, which is a major limitation for long-running activities. As illustrated in Figure 5, the Lambda processing time in ABAC is significantly lower than in ACAC because ACAC continuously evaluates policies throughout the activity's lifecycle, whereas ABAC makes a one-time access decision without updates. Another critical metric in our evaluation was a number of completed full-cycle activities, which includes pre-, ongoing-, and post-phases of an activity life cycle. As observed in Figure 6, the ABAC model fails to complete the full cycle for any number of requests, whereas ACAC successfully completes a substantial number of activity full cycles. Additionally, the number of initial requests satisfied (allowing the activity to execute) before execution is significantly lower in ABAC, primarily due to (i) object unavailability in the system and (ii) missing update mechanisms to ensure dependent activities reach their desired states. These findings emphasize the applicability of ACAC in environments where continuous evaluation and dynamic activity execution cycles are required. Overall, the performance evaluation demonstrates that the ACAC model provides a robust, scalable, and dynamic access control solution that effectively manages complex activity dependencies in highly dynamic environments. By leveraging cloud-based services for proof-of-concept implementation, we validate the model's feasibility in real-world IoT ecosystems. The use of multithreading for concurrent request handling and the automated update process for dependent activities further reinforce the practicality and efficiency of ACAC. These results strongly support ACAC as a superior alternative to traditional access control models for large-scale, smart and dynamically changing connected IoT environments.

6. Conclusion

In this paper, we presented a comprehensive implementation of the $\mathrm{ACD}_{\mathrm{ACAC}_{\mathrm{ABCD}}}$ model, demonstrating its feasibility in dynamic and interconnected smart environments. By incorporating all decision parameters — Authorizations, Obligations, Conditions, and Dependencies—along with constraints, we addressed critical challenges in managing long-lived IoT device operations while ensuring security and policy compliance. We implemented the model using Amazon Web Services (AWS) and evaluated its performance against the widely used ABAC model in a smart farming use case. Our results indicate ACAC

efficiently handles complex dependencies, supports continuous policy enforcement, and scales effectively under increasing workloads. Unlike ABAC, which lacks ongoing access control, ACAC ensures secure and adaptable decision-making throughout an activity's lifecycle. These findings validate the practicality and scalability of ACAC in real-world IoT ecosystems, making it a promising approach for securing and managing dynamic, interconnected smart systems. Future research may explore further optimizations and real-time implementation in broader IoT domains.

ACKNOWLEDGEMENT
This work is partially supported by the National Science Foundation grants 2416990 and 2346001.

REFERENCES

[1] Deborah D Downs, Jerzy R Rub, Kenneth C Kung, and Carole S Jordan. Issues in discretionary access control. In *1985 IEEE symposium on security and privacy*, pages 208–208.

[2] Yanfang Fan, Zhen Han, Jiqiang Liu, and Yong Zhao. A mandatory access control model with enhanced flexibility. In *2009 international conference on multimedia information networking and security*, volume 1, pages 120–124. IEEE, 2009.

[3] Ravi S Sandhu. Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier, 1998.

[4] Xin Jin, Ram Krishnan, and Ravi Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *Data and Applications Security and Privacy XXVI: 26th Annual IFIP WG 11.3 Conference, DBSec 2012, Paris, France, July 11-13, 2012. Proceedings 26*, pages 41–55. Springer, 2012.

[5] Maanak Gupta and Ravi Sandhu. Towards activity-centric access control for smart collaborative ecosystems. In *Proceedings of the 26th ACM symposium on access control models and technologies*, pages 155–164, 2021.

[6] Tanjila Mawla, Maanak Gupta, and Ravi Sandhu. BlueSky: Activity Control: A Vision for Active Security Models for Smart Collaborative Systems. In *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*, pages 207–216, 2022.

[7] Tanjila Mawla, Maanak Gupta, Safwa Ameer, and Ravi Sandhu. The $\mathrm{ACAC_D}$ model for mutable activity control and chain of dependencies in smart and connected systems. *International Journal of Information Security*, 23(5):3283–3310, 2024.

[8] Tanjila Mawla, Maanak Gupta, and Ravi Sandhu. Specification and Enforcement of Activity Dependency Policies using XACML. In *2024 10th IEEE International Symposium on System Security, Safety, and Reliability (ISSSR)*, pages 429–440.

[9] Tanjila Mawla and Maanak Gupta. Constraints Visualization and Specification for Activity-centric Access Control. In *2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*, pages 371–380. IEEE, 2024.

[10] Xuanxia Yao, Fadi Farha, Rongyang Li, Ismini Psychoula, Liming Chen, and Huansheng Ning. Security and privacy issues of physical objects in the IoT: Challenges and opportunities. *Digital Communications and Networks*, 2021.

[11] Leonardo Babun et al. A survey on IoT platforms: Communication, security, and privacy perspectives. *Computer Networks*, 192:108040, 2021.

[12] Shoubai Nie, Jingjing Ren, Rui Wu, Pengchong Han, Zhaoyang Han, and Wei Wan. Zero-Trust Access Control Mechanism Based on Blockchain and Inner-Product Encryption in the Internet of Things in a 6G Environment. *Sensors*, 25(2):550, 2025.

[13] Chen Zhonghua, SB Goyal, and Anand Singh Rajawat. Smart contracts attribute-based access control model for security & privacy of IoT system using blockchain and edge computing. *The Journal of Supercomputing*, 80(2):1396–1425, 2024.

[14] Rudri Kalaria, ASM Kayes, Wenny Rahayu, Eric Pardede, and Ahmad Salehi Shahraki. Adaptive context-aware access control for IoT environments leveraging fog computing. *International Journal of Information Security*, 23(4):3089–3107, 2024.

[15] Maanak Gupta, Feras M Awaysheh, James Benson, Mamoun Alazab, Farhan Patwa, and Ravi Sandhu. An attribute-based access control for cloud enabled industrial smart vehicles. *IEEE Transactions on Industrial Informatics*, 17(6):4288–4297, 2020.

[16] Safwa Ameer, Lopamudra Praharaj, Ravi Sandhu, Smriti Bhatt, and Maanak Gupta. ZTA-IoT: A Novel Architecture for Zero-Trust in IoT Systems and an Ensuing Usage Control Model. *ACM Transactions on Privacy and Security*, 2024.

[17] Maanak Gupta, Mahmoud Abdelsalam, Sajad Khorsandroo, and Sudip Mittal. Security and privacy in smart farming: Challenges and opportunities. *IEEE access*, 8:34564–34584, 2020.

[18] Safwa Ameer, James Benson, and Ravi Sandhu. Hybrid approaches (ABAC and RBAC) toward secure access control in smart home IoT. *IEEE transactions on dependable and secure computing*, 20(5):4032–4051, 2022.

[19] Philip WL Fong and Ida Siahaan. Relationship-based access control policies and their policy languages. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, pages 51–60, 2011.

[20] Jaehong Park and Ravi Sandhu. The $\mathrm{UCON_{ABC}}$ usage control model. *ACM transactions on information and system security (TISSEC)*, 7(1):128–174, 2004.

[21] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero Trust Architecture. *National Institute of Standards and Technology*, 2020.