Effective Handling of Erroneous Executions of Automation Rules on Multi-Platform Smart Home IoT Systems

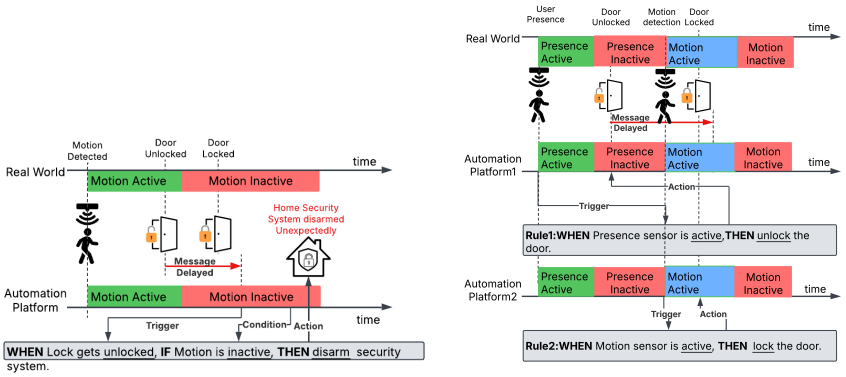
Zhuoyi Shi[1][0009−0007−3080−3604], Chenglong Fu[2][0000−0001−6555−9858], Xiaojiang Du[1][0000−0003−4235−9671]✉, and Rong Xu[1][0009−0002−2942−7086]

[1] Stevens Institute of Technology, Hoboken, NJ, USA
zshi23@stevens.edu, ✉ xdu16@stevens.edu, rxu35@stevens.edu
[2] University of North Carolina at Charlotte, Charlotte, NC, USA
chenglong.fu@charlotte.edu

Abstract. Smart-home automation systems are driven by events from IoT devices, yet varying network delays across IoT platforms can reorder events and cause problems. Latency gaps across platforms amplify the problem, causing actions to be executed inconsistently and sometimes leaving devices and homes in unsafe states. When rules span multiple IoT platforms with divergent views of device status, various problems arise. Recent work has found that latency can be exploited by attackers to trigger automation rules erroneously or disable their execution. In this paper, we systematically analyze erroneous automation-rule executions and identify inconsistent execution contexts across automation rules on different IoT platforms as their root cause. To address this problem, we present **IoT-Handler**, a local event mediator and state synchronizer designed to ensure temporal and logical consistency across platforms through a novel execution-context isolation mechanism. **IoT-Handler** intercepts device communications via virtual device proxies, decoupling devices from direct cloud interactions. Each automation trigger is encapsulated in an isolated execution context that captures relevant device states and conditions. IoT-Handler also employs a selective queuing strategy to serialize conflicting rules while allowing independent rules to run in parallel. Additionally, it synchronizes virtual and physical device states using a controlled, selective freezing mechanism to prevent race conditions. We conduct an extensive evaluation using multiple IoT platforms with deliberate timing disruptions, and our real-world experiments demonstrate that **IoT-Handler** effectively prevents unsafe and inconsistent device behaviors caused by out-of-order event processing. Unlike prior solutions, **IoT-Handler** achieves cross-platform synchronization and robust automation without requiring modifications to existing devices or automation logic on IoT cloud platforms.

Keywords: Smart Home · Message Delay · Internet of Things

1 Introduction



(a) Disordered single-platform automation-rule execution caused by IoT message delay.

(b) Disordered cross-platform automation-rule execution caused by IoT message delay.

Fig. 1. Examples of interaction threats.

The rapid development of the Internet of Things (IoT) has led to the proliferation of smart home systems. Recent statistics indicate that approximately 45% of U.S. households with internet access possess at least one smart home device [25]. This widespread adoption is enabled by IoT platforms such as Amazon Alexa, Google Home, and Apple HomeKit, which allow users to automate devices via trigger–condition–action (TCA) rules [26]. Figure 1 illustrates two rules configured across different platforms: (1) *R1*: WHEN user presence becomes active, THEN unlock the door; and (2) *R2*: WHEN the porch motion sensor is active, THEN lock the door. Intuitively, R1 unlocks the door upon the user's return, while R2 enforces a locking action during departure (e.g., a deliberate exit gesture captured by the porch sensor). For instance, as shown in Figure 1, if a user briefly returns home and then leaves again, an attacker could maliciously delay either the "Presence On" event or the "Door Unlock" command. This delay would postpone the execution of Rule 1 until after Rule 2 (triggered by the user's departure) has already executed. As a result, the delayed command from Rule 1 would cause the door to remain unlocked while the user is away, significantly increasing the risk of burglary.

Existing intrusion and anomaly detection systems in IoT environments primarily focus on identifying malicious or conflicting logic across automation rules, including cross-app interference among individually benign automation apps [8], or on detecting the injection of unauthorized events and commands [14]. However, these systems do not take the event latency into consideration and cannot handle the issues that arise from the misordering of legitimate IoT events.

Xu et al. [28] proposed a method to detect stealthy delay attacks in IoT systems. However, that work focuses on detecting attacks rather than preventing or mitigating them. Furthermore, it considers only a single-platform environment and thus cannot handle inconsistencies across multiple IoT platforms. Consequently, ensuring correct execution ordering of automation rules in multi-platform IoT ecosystems remains an open problem.

In this paper, we present IoT-Handler, a systematic approach to addressing delay-induced erroneous rule executions in smart homes. *IoT-Handler* is a local event mediator designed to handle these execution-order issues without replacing existing IoT cloud platforms. In fact, prior IoT security papers at major security conferences also use local mediator devices, although those works address different threats.

Through analysis of various delay attack scenarios, we identify the root cause as violations of a rule's execution context, which is the snapshot of relevant device states at the moment of rule triggering. Our key insight is that by strategically controlling state propagation between IoT devices and platforms, we can prevent event reordering and the resulting conflicting actions. IoT-Handler achieves this by decoupling direct device-to-platform connections and instead routing all communications through virtual device proxies that represent each physical device on each platform. Through careful state synchronization between physical devices and their virtual counterparts, IoT-Handler maintains precise control over event delivery timing and ordering. This architecture ensures that all platforms observe a consistent sequence of events that accurately reflects physical device states, enabling automation rules to execute reliably and correctly across heterogeneous platforms.

Building upon this insight, we further enhance system efficiency and user experience by eliminating unnecessary latencies introduced by indiscriminately enforcing a sequential order on all IoT events. Consider an additional automation rule, **Rule 3**: *WHEN smoke is detected, THEN sound an alarm and unlock the door*, deployed alongside the two rules depicted in Figure 1. In attack scenarios where the execution of Rule 1 is delayed, a naive approach that strictly enforces sequential event ordering would inadvertently postpone the execution of Rule 3 until the delay attack ends. Such unnecessary delays could significantly impede critical safety actions triggered by Rule 3 if smoke is detected during the attack. To address this shortcoming, we propose a selective *Event Queue Mechanism* that intelligently identifies and queues only conflicting events, allowing independent events to execute immediately. This refined strategy maintains essential logical ordering for correctness and safety, substantially reducing worst-case latency from minutes to seconds, as confirmed by our experimental evaluation.

We implement IoT-Handler as a lightweight mediator deployable on commodity hardware (e.g., a Raspberry Pi running OpenWrt). In a real apartment with 20 heterogeneous devices across three platforms (Amazon Alexa [1], Google Home [16], Apple HomeKit [2]) and 20 automation rules, IoT-Handler prevents race conditions and state inconsistencies observed in unmediated baselines. Under normal operation, it adds 0.2–0.5 s overhead; during conflict resolution, 5–

10 s delays remain acceptable for safety-critical actions (e.g., locks). Across eight representative conflict scenarios (C1–C8; see Section 5), IoT-Handler achieved correct outcomes in all cases while preserving concurrency for independent rules.

In summary, our contributions are as follows:

- **Systematic Analysis of Delay-Induced Failures**: We perform a comprehensive analysis of erroneous automation executions caused by stealthy delay attacks in multi-platform IoT environments. We formally model an automation rule's execution context and identify two fundamental ways delays cause errors: (i) an inconsistent context at trigger time, and (ii) changes to an ongoing execution context.
- **Context Encapsulation.** We propose a novel execution-context encapsulation mechanism that creates isolated virtual device instances for each automation rule. This approach ensures temporal and logical consistency by controlling state synchronization between physical devices and their virtual counterparts, effectively preventing cross-platform race conditions.
- **Selective Queuing for Performance Optimization**: We design an intelligent event queue mechanism that distinguishes between dependent and independent automation events. By selectively queuing only conflicting events while allowing concurrent executions of unrelated rules, we significantly improve system responsiveness and user experience, reducing unnecessary delays from minutes to seconds.
- **Real-World Implementation and Comprehensive Evaluation**: We implement IoT-Handler on commodity hardware and evaluate it in a real-world multi-platform smart home with 20 devices and 20 automation rules. Our experiments demonstrate that IoT-Handler achieves 100% correctness in preventing race conditions with minimal overhead (0.2–0.5 s), while successfully defending against delay attacks or large delays due to network issues.

The rest of this paper is organized as follows. Section 2 defines the problem scope and threat model. Section 3 describes the overall system model and adversarial threat-model assumptions. Section 4 details our implementation of IoT-Handler. Section 5 evaluates its correctness and performance through extensive experiments. Section 6 reviews related work, and Section 7 concludes the paper. Appendix A describes the ethical considerations and safeguards, and Appendix B provides real-world examples of the IoT automation tests.

2 Background

2.1 Smart Home Systems

Modern smart home platforms typically consist of three main components: (i) IoT devices (sensors and actuators deployed in the home), (ii) user interfaces (mobile or web applications and voice assistants), and (iii) cloud services [18,31]. The cloud service plays a central role in coordinating communication among devices and control channels. IoT devices report sensor readings or state changes

to the cloud (often via a local hub or directly over Wi-Fi), and the cloud relays commands to actuator devices, ensuring that all components maintain a consistent view of the home's state.

To automate device behaviors, smart home platforms allow users to define trigger–condition–action (TCA) rules that execute on the cloud servers. In this paradigm, a rule specifies a trigger event (such as a sensor reading or device state change), an optional condition predicate, and an action to perform when the trigger occurs and the condition holds. The cloud backend continuously listens for device events that match rule triggers; when a trigger event is detected, the platform evaluates any rule conditions (for example, time of day or another device's status) before executing the associated action on the target device. Such automation rules are managed and executed by the cloud service, which maintains the necessary state and logic for evaluating triggers and conditions.

2.2 IoT Message Delay Attacks

Recent work by Fu et al. [13] discovered the stealthy message delay attack, which can affect all IoT events and command messages in smart home IoT systems. Unlike brute-force methods such as wireless jamming that typically result in message loss (undelivered messages), the stealthy delay attack ensures delayed messages are still successfully delivered and accepted by home automation platforms, as it does not trigger protocol timeouts. By selectively delaying messages from certain IoT devices while permitting others to proceed normally, an attacker can intentionally alter the sequence in which IoT messages arrive at an automation platform. This capability exposes home automation systems to attacks such as the *Delay-based Automation Interference* attack [5] and the *Erroneous Execution* attack [13]. For instance, consider an automation rule: "When motion is active, if the door is closed, then open the window." Under normal conditions, this rule would not trigger when a user leaves the room and closes the door, as the "motion-active" event precedes the "door-closed" event. However, by selectively delaying the "motion-active" message so that it arrives after the "door-closed" event, an attacker can maliciously trigger the execution of this rule after the user departs. Consequently, the window would unexpectedly open after the user has left, creating potential safety risks.

3 System and Threat Models

3.1 System Model

We consider a typical smart home environment, as illustrated in Figure 2, where IoT devices are integrated with multiple heterogeneous IoT platforms. Such multi-platform deployments are increasingly common among smart home users, as a recent study [5] indicates. In our model, IoT devices connect through a home Wi-Fi router and establish TCP/IP connections to their respective cloud

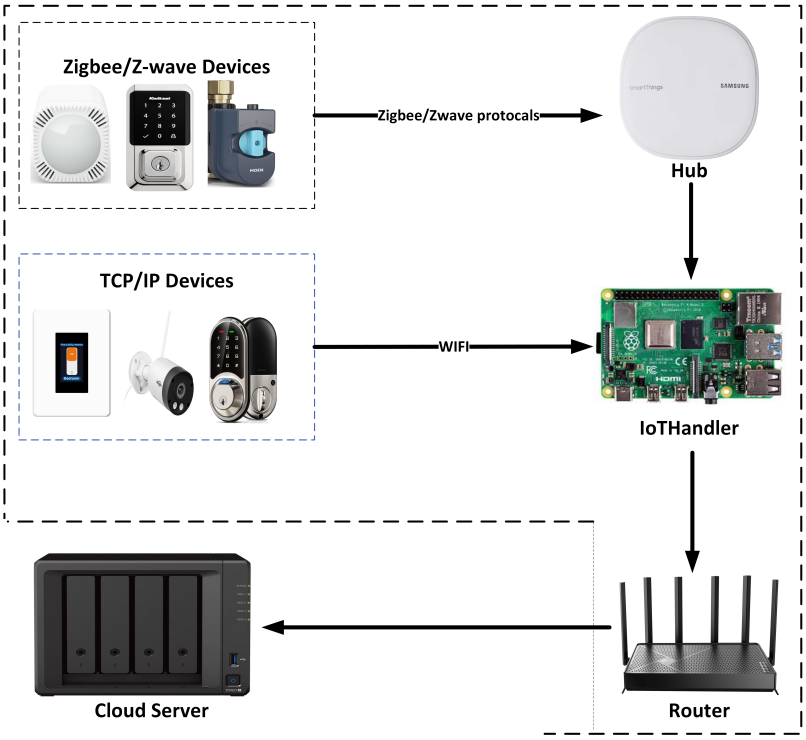


Fig. 2. Architecture of the IoT-Handler system model.

platform servers. Devices that lack native TCP/IP support (e.g., Zigbee or Z-Wave sensors) communicate via IoT hubs that act as border routers, bridging those protocols onto the home network.

As documented in [5], users frequently deploy automation rules that span multiple IoT platforms. These automation rules trigger and execute based on device state observations maintained by their hosting platforms, which are updated through events transmitted by IoT devices. Due to inherent variations in network latency across different device-to-platform connections, multiple platforms may hold inconsistent views of the same device's state at any given moment. This temporal inconsistency forms the foundation of the security vulnerabilities we address in this work.

To mitigate these vulnerabilities, we deploy IoT-Handler on a local computing device (such as a Raspberry Pi) positioned between the IoT devices/hubs and the home Wi-Fi router. IoT-Handler operates as a local event mediator and state synchronizer. It does not execute rule logic itself. **This local mediator architecture is similar to state-of-the-art approaches published at major security conferences [14].** For instance, IoTMediator [6] employs

a similar local mediator model to detect and handle interaction threats across multiple control channels. IoT-Handler intercepts events from IoT devices, ensures that they are delivered to the respective cloud platforms in a consistent and chronologically correct order, and mediates the resulting action commands. All trigger–condition–action automation rules are executed by the original IoT cloud platforms (e.g., Amazon Alexa and Google Home). The focus of this paper is on solving delay-induced inconsistencies through context encapsulation.

3.2 Threat Model

In this paper, we consider an attacker whose goal is to disrupt or manipulate the execution of smart home automation rules using the stealthy delay attack discussed in [5,13]. We adopt the same attacker model as these prior works, in which attackers can eavesdrop on and selectively delay encrypted traffic between IoT devices/hubs and IoT cloud platforms. The attacker achieves this through network attack primitives such as wireless packet capture, ARP spoofing, or TAP (Test Access Point) attacks on the user's home Ethernet cable.

Consistent with assumptions in related IoT defense systems such as IoT-Guard [4], IoTSafe [10], and MP-Mediator [28], we assume that IoT-Handler itself remains secure (i.e., not compromised). Furthermore, we assume the device hosting IoT-Handler is protected against ARP spoofing attacks using some existing defense mechanisms such as ArpON [3], XArp [27], and static ARP entries [22]. Consequently, the communication channel between IoT devices and IoT-Handler is considered secure and immune to delay attacks, allowing IoT-Handler to maintain an accurate real-time view of device states.

4 Design of IoT-Handler

To systematically address the safety issues arising from out-of-order IoT messages, we propose **IoT-Handler**, a locally hosted event-reordering service designed to ensure the logical correctness of automation rules, even when they are deployed across different automation platforms. In this section, we first introduce the concept of an IoT automation execution context and utilize this concept to analyze the root cause of erroneous automation executions. Based on this analysis, we then propose a countermeasure that encapsulates the execution contexts of automation rules that are potentially vulnerable to message delay attacks. Consider again the scenario from Figure 1: a homeowner has two automation rules across two platforms for securing the home upon departure.

- Rule 1 (Platform A): WHEN user presence becomes active, THEN unlock the door.
- Rule 2 (Platform B): WHEN the porch motion sensor is active, THEN lock the door.

Rule 1 automatically unlocks the door when the user comes home, and the porch-mounted motion sensor in Rule 2 detects the user's deliberate exit gesture,

ensuring that the door is locked upon departure. If a user briefly returns home and then leaves, an attacker could delay either the presence "On" event or the door-unlock command. As a result, Rule 1's execution would be deferred until after Rule 2 has run, leaving the door unlocked while the user is away. This outcome, as discussed earlier, significantly increases the risk of unauthorized entry.

4.1 Automation Rule Execution Context

In this subsection, we formalize the notion of an execution context, which we use to model the logical correctness of automation-rule execution. Based on the formalization, we then show that every delay-induced error can be interpreted as a violation of one of two assertions about this execution context.

Let R be an automation rule and let $Dep(R)$ denote the set of devices on which R depends (i.e., any device referenced as R's trigger, condition, or action). We denote the state of device d at time t by $\mathcal{S}(d,t)$. The execution context of R at trigger time t is defined as

$$\mathbb{C}_R(t) = \{\mathcal{S}(d,t) \mid d \in Dep(R)\} \tag{1}$$

Here, $\mathcal{S}(d,t)$ stands for the device's state in the physical world, and $\mathbb{C}_R(t)$ captures the *ground-truth* execution context on which rule R relies at the triggering moment. For an automation rule to execute correctly, two assertions about the execution context must hold:

- It should be consistent with the ground truth.
- It should remain immutable during the execution period (i.e., from the occurrence of the trigger event until the action is executed).

After the platform receives the status of the related devices (e.g., triggers and conditions), it executes an automation rule quickly because it needs to run only a few lines of code.

These assertions hold under normal operating conditions, in which the transmission of IoT messages and the execution of automation rules take negligible time. When message delays occur, however, both rule execution and state propagation can be extended by tens of seconds to several minutes (as measured by Fu et al. [13]). In such cases, logical errors arise precisely when one of the two assertions is violated. Indeed, every delay-induced error reported in [5,13,15] can be mapped to the violation of at least one assertion above. Below, we illustrate these two classes of erroneous execution with examples.

Execution with inconsistent context. If, at trigger time, any device state in the execution context is inconsistent with the ground truth, the rule may evaluate its conditions incorrectly. Consider a common security rule: "When the motion sensor is active and presence is off, issue an alarm notification." An attacker may break in while delaying the event $\mathcal{E}^{\text{presence}}_{\text{on}}$ generated when the user leaves. When the rule is triggered by $\mathcal{E}^{\text{motion}}_{\text{active}}$, the presence-sensor state $\mathcal{S}^{\text{presence}}_{\text{on}}$ recorded in the context differs from the ground truth. Executing the rule under

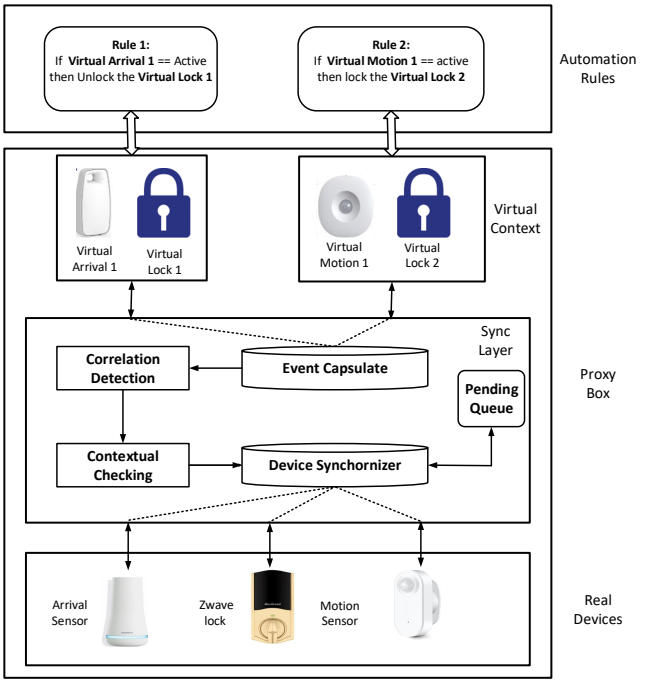


Fig. 3. Illustration of context encapsulation for the two example automation rules in Figure 1.

this inconsistent execution context violates the first assertion. As a result, the alarm is muted. Moreover, as long as the attacker maintains active motion in front of the motion sensor, the sensor will remain active and not produce a new $\mathcal{E}^{\text{motion}}_{\text{active}}$ event. Therefore, the rule will not have another chance to execute, even after the delayed presence event arrives.

Changes to an ongoing execution context. Message delays can also prolong a rule's execution period if a trigger or action message is delayed. If, during this extended execution window, another rule modifies any device state in the rule's execution context, the second assertion is violated. As illustrated in Figure 1, rule R_1 is triggered at t_1, producing the context $\mathbb{C}_{R_1}(t_1) = \{\mathcal{S}^{\text{presence}}_{\text{off}}, \mathcal{S}^{\text{lock}}_{\text{locked}}\}$. This context should remain unchanged until command $\mathcal{C}^{\text{lock}}_{\text{unlocked}}$ completes. However, because event $\mathcal{E}^{\text{presence}}_{\text{on}}$ is delayed, the execution of R_1 is halted in a pending state. Meanwhile, rule R_2 sets the lock state to $\mathcal{C}^{\text{lock}}_{\text{locked}}$. This state change violates R_1's immutability assertion and explains the observed erroneous behavior.

4.2 Execution Context Encapsulation

Based on the preceding analysis, we propose a countermeasure that temporarily suspends state synchronization between physical devices and their virtual counterparts to preserve the chronological ordering of events across all platforms.

Specifically, we delay the propagation of an event until all preceding events have been successfully acknowledged by their associated IoT platforms, ensuring that every platform observes events in the same temporal sequence as their physical occurrence. For example, in Figure 1, we postpone the synchronization of the "Presence active" event (which triggers R_1) until R_2 completes its execution, preventing R_2's premature lock command from overriding R_1's intended unlock action.

However, implementing this countermeasure in multi-platform smart home deployments presents challenges. When a single physical device participates in rules across different platforms, their execution contexts inevitably overlap, yet each platform maintains only a partial view of the system state. This architectural limitation makes race conditions inevitable whenever rules from different platforms share devices. For example, when rules R_1 and R_2 are deployed on Platforms A and B, respectively, and operate on the same lock, Platform B cannot detect that R_1's execution is ongoing and thus cannot prevent its conflicting command from interfering with R_1's context.

To address the challenges posed by overlapping execution contexts, we propose context encapsulation, a method that isolates the execution context of each automation rule. Specifically, for every physical IoT device, we instantiate a set of virtual device instances, creating one instance for each associated automation rule. Consequently, each rule operates exclusively on its dedicated set of virtual devices, effectively eliminating context overlaps. All virtual device instances are hosted locally by IoT-Handler, which (i) maintains a global view of device states across rules, even if the rules cross different platforms, and (ii) exerts fine-grained control over synchronization between physical devices and their virtual counterparts. As depicted in Figure 3, IoT-Handler creates virtual instances for each of the three devices involved in the two automation rules. Notably, it generates two distinct virtual instances for the same door lock, dedicated individually to Rule 1 and Rule 2. Thus, even if the two rules are deployed on separate platforms, IoT-Handler can precisely orchestrate their execution order by carefully managing state synchronization between physical and virtual devices.

When a trigger event occurs, IoT-Handler identifies the affected rules and snapshots the states of the devices in their dependency sets. It then temporarily freezes synchronization for only those devices. Platforms interact only with the corresponding virtual devices; commands are forwarded to the physical device and are considered completed when the device emits a confirming (feedback) event. This design gives IoT-Handler a global view of shared devices while preventing one rule's delayed execution from affecting another rule's context.

This workflow ensures that while $\mathbb{C}_R(t)$ is active, devices D_R behave as if time were "frozen" at t from R's viewpoint. The rule executes with full isolation: no other rule can interfere with D_R, and R does not observe any state changes in D_R beyond t until it completes. When the context capsule terminates and the freeze is lifted, the system transitions back to real-time operation by processing any queued events, thereby preserving overall system reactivity without compromising the consistency of R's execution.

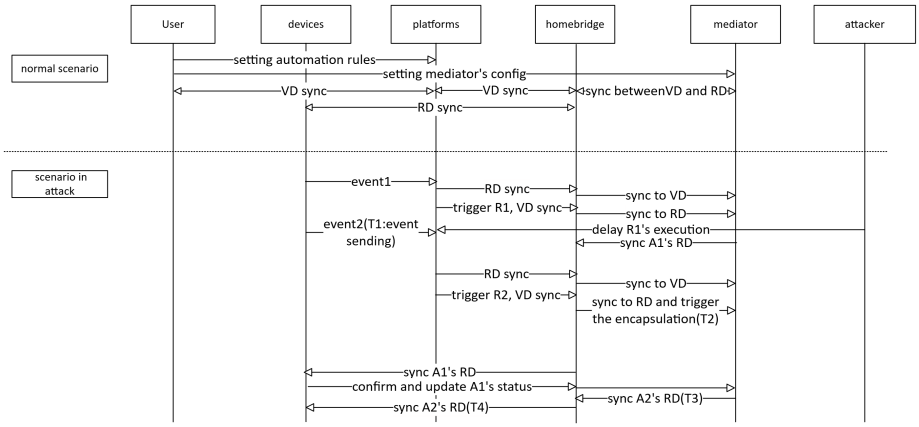


Fig. 4. Time-sequence diagram of IoT-Handler mediating two overlapping automation rules.

This robust freeze/resume functionality is realized by the IoT-Handler's internal DeviceSynchronizer component, a unified control layer designed specifically to orchestrate device interactions and uphold temporal consistency. DeviceSynchronizer has the ability to temporarily halt (pause) updates and subsequently resume device synchronization precisely when a given execution context requires controlled ordering. Concretely, whenever a rule is activated, the handler—via the DeviceSynchronizer—marks the devices directly involved as temporarily locked to the executing context, meaning any further commands or incoming events targeting these specific devices are held back until the current automation rule completes its actions. Upon action completion, the previously locked devices are promptly released, allowing normal message handling to resume. Critically, this locking mechanism is applied selectively rather than globally: only the devices directly engaged by the executing rule are frozen, while unrelated devices across the smart home remain fully operational and responsive. This selective freezing significantly improves overall system performance and responsiveness, allowing unrelated automations, such as lighting controls or speaker adjustments, to run concurrently without interference, while carefully serializing interactions involving shared devices to guarantee correctness.

In more detail, IoT-Handler combines context capsules and temporal scheduling to enforce ordered yet efficient rule execution. Figure 4 presents a time-sequence scenario involving two overlapping automation rules coordinated by IoT-Handler. T_1 denotes the initial trigger (device-event generation) for Rule 1 on Platform A, which IoT-Handler captures at T_2. Shortly thereafter, a second trigger for Rule 2 on Platform B is detected. IoT-Handler forwards the first trigger to Platform A, which returns an action command at time T_3. Because Rule 1 and Rule 2 target the same device (a door lock), IoT-Handler imposes a brief freeze on Rule 2's command to prevent conflicts. Rule 1's command is

Algorithm 1: Isolated Execution with Device Freezing for Rule R.

Input: Trigger event e on device d_{trig} at time t, rule R
Output: Rule R executes on the snapshot of its devices

```
begin
    D_R ← all sensors and actuators used by R; C_R(t) ← S(d,t) | d ∈ D_R;
     freeze(D_R) ;        // lock devices in D_R and buffer their events
    foreach d ∈ D_R do
        begin
            buffer incoming events for d; ;    // hold events until unfreeze
    execute rule R using context C_R(t); unfreeze(D_R) ;  // unlock devices
     in D_R
    foreach d ∈ D_R do
        begin
            deliver buffered events for d in original arrival order;
```

dispatched to the physical device and completes at T_4, after which the queued Rule 2 command is released at T_3' and executed, reaching completion at T_4'. This controlled freeze/resume scheduling (highlighted by the dashed pause) ensures serialized execution and avoids device-state conflicts. Each rule executes within its isolated context capsule and references the stable state at its respective T_1, while IoT-Handler synchronizes the final device states across Platforms A and B after the actions to maintain consistency. When a triggering event (T_1) occurs on a physical device (e.g., a motion sensor detects activity or a door lock engages), IoT-Handler immediately intercepts the event through the device's virtual proxy, records an accurate timestamp (T_2), and establishes context capsules for each automation rule activated by the trigger. Each capsule captures a snapshot of relevant device states at the precise trigger moment, isolating the rule's context so that, despite subsequent events, each platform evaluates its automation logic using a consistent view of the environment. The event is then forwarded to the corresponding cloud platform, which interacts exclusively through the virtual device proxies and evaluates and issues commands as usual.

4.3 Context-Aware Message Handling

A key design principle of IoT-Handler is to reduce latency by avoiding unnecessary queuing and delays. Our system does not queue all events. Instead, it performs a dependency analysis for each triggered rule, checking for resource conflicts (e.g., two rules targeting the same actuator). The second rule's execution is temporarily enqueued only when a direct conflict is identified. Independent rules are processed concurrently without delay. This selective approach ensures safety and correctness where required while preserving system responsiveness for non-conflicting operations.

IoT-Handler queues only conflicting events, which is illustrated in Figure 5. For each trigger, it performs a dependency check over the devices referenced

by the triggered rules. If no locked device is involved, the event is immediately propagated to the relevant virtual devices and cloud platforms. If the event conflicts with an active context, it is inserted into a PendingQueue and released when the earlier context completes. Queued events are processed in FIFO order per device or dependency group, which preserves local arrival order while still allowing unrelated rules to execute concurrently. To prevent stale commands from reverting newer physical states, IoT-Handler also tracks each device's latest action timestamp and discards or overrides commands derived from older triggering contexts. Together, dependency checks, selective freezing, and timestamp validations ensure ordered execution for dependent rules while preserving concurrency for independent ones.

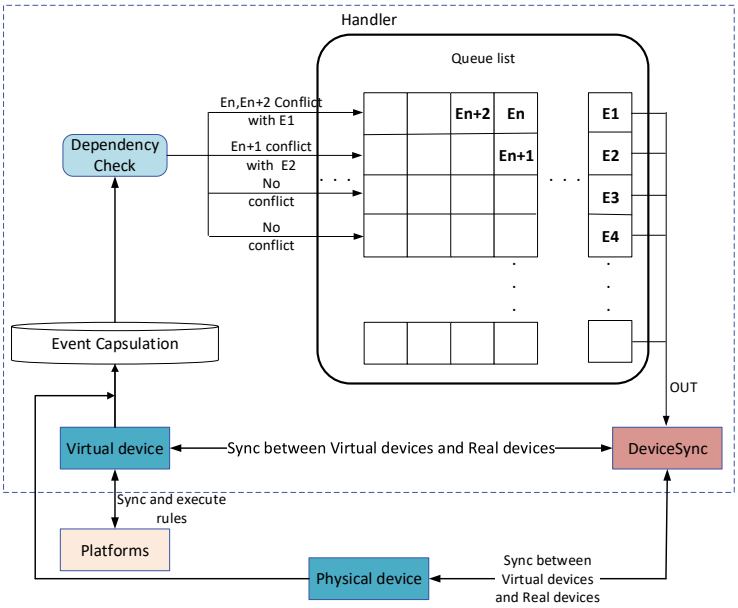


Fig. 5. IoT-Handler's internal message-handling workflow and PendingQueue mechanism.

Message Handling All IoT device messages—such as sensor readings and actuator status updates—are first intercepted by IoT-Handler's internal modules as they occur. When a device event is captured, the handler immediately timestamps and logs it (denoted as time T_2, representing when the handler received the event, relative to the device's original event time T_1). Next, the handler performs a dependency check to determine whether the source device, or any other device implicated by the event's corresponding automation rule, is currently involved in an ongoing execution. If the check finds no conflicts (i.e., none of the

relevant devices are locked by an active rule), the event can be processed without delay. In that case, IoT-Handler updates the state of the corresponding virtual device(s) and promptly forwards the event notification to the appropriate cloud platform(s), triggering the associated automation rules as normal. Simultaneously, the handler intercepts any action commands issued by the platforms in response (timestamping these at time T_3, when the handler dispatches them) and mediates the communication to the physical devices.

If the dependency check reveals that a new event involves a device currently frozen (locked in another rule's active context capsule), IoT-Handler delays processing that event. Instead of forwarding it to the cloud immediately, the event is placed in a dedicated PendingQueue. In effect, the new trigger is frozen in the queue until it can be safely processed in the correct sequence. The handler dynamically decides when to release events from the queue based on the progression of rule contexts. Once an ongoing rule completes (indicated by its timestamp T_4, confirming that its device updates are complete), IoT-Handler unlocks the devices involved in that rule. At that point, queued events waiting on those devices are dequeued and processed in chronological order. This ensures that logically dependent events execute sequentially, while independent events bypass unnecessary waiting and execute concurrently.

Beyond dependency-driven freezing, IoT-Handler implements a timestamp-based arbitration mechanism to prevent outdated or stale commands from violating temporal consistency. Specifically, the handler maintains a timestamp for the most recent state change of each physical device, called the device's latest action time (LAT). When a platform sends a new command to a device through the handler, this LAT is compared with the triggering-event timestamp of the incoming command. If the incoming command references an older event (for example, the device has undergone a more recent change), the handler recognizes the command as outdated and either discards or overrides it, preventing reversions to earlier states that could otherwise arise from network-induced delays or late arrivals. Through this combination of dependency checks, selective freezing, and timestamp validation, IoT-Handler ensures that automation rules execute within their correct contextual state and in the proper chronological order, thereby maintaining overall correctness and consistency.

Figure 5 visually illustrates the message handling and synchronization workflow within the IoT-Handler, highlighting how incoming events are managed, context-checked, queued, and ultimately dispatched to physical devices and external platforms.

PendingQueue Mechanism To manage the concurrency of multiple automation rules, IoT-Handler uses a structured PendingQueue. Whenever an event triggers an automation rule, IoT-Handler immediately evaluates rule dependencies. Rules that share triggers, conditions, or actions are grouped into shared queues to enforce sequential execution, while unrelated rules occupy distinct queues that allow parallel execution. This fine-grained queue management ensures that only

dependent interactions incur delays, while independent ones proceed without hindrance, thus maximizing concurrency and throughput.

In practice, PendingQueue comprises several specialized queues, each assigned either to an individual device or to a group of interdependent devices. If an incoming event or command is delayed because of an active context (e.g., if a device is locked by an ongoing automation execution), it is immediately placed in the corresponding PendingQueue with its triggers and actions. These queues operate on a first-in, first-out basis, preserving local arrival order and ensuring that events related to the same device context execute in strict sequential order.

5 Evaluation

We evaluate IoT-Handler's performance in a representative multi-platform smart home environment. First, we detail our experimental setup, including the physical layout, the IoT devices and platforms used, and the set of automation rules (Section 5.1). Next, we present case studies that demonstrate IoT-Handler's correctness in preventing rule conflicts under deliberate message delays (Section 5.2). Finally, we measure the end-to-end latency of IoT-Handler under both normal conditions and stress-test scenarios with induced delays (Section 5.3).

5.1 Experimental Setup and Methodology

Physical Layout We begin by detailing our experimental setup, which represents the complex conditions of modern multi-platform smart homes. As shown in Figure 6, we deployed IoT-Handler, several IoT devices, and automation rules across multiple platforms in a real-world studio apartment to test its behavior under both natural and synthetic smart-home scenarios.

The testbed included a single-resident studio apartment with typical smart-home zones: a bedroom, living room, and entrance corridor. In this environment, we installed motion sensors, light bulbs, switches, locks, and other automation-capable devices commonly found in commercial homes. The deployment was functional and continuous; we recorded both natural resident behavior and scripted interactions, with Institutional Review Board (IRB) approval in place for scenarios involving safety-critical devices such as smart locks or heating units.

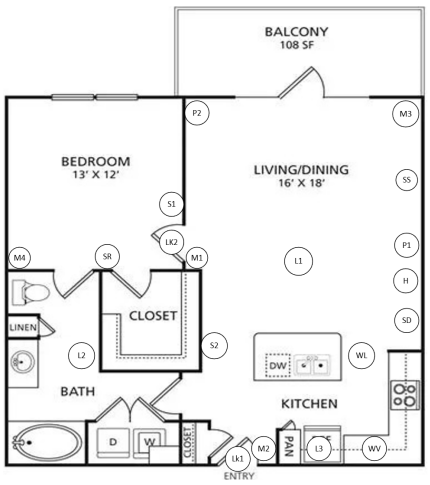


Fig. 6. Floor plan of the studio testbed.

Device Overview Table 1 lists the 20 IoT devices incorporated into the ex-

periment, along with their wireless protocols, principal attributes, and integration platforms. Most devices rely on Wi-Fi, Zigbee, or Z-Wave networks and connect to cloud services such as Alexa, Google Home, or HomeKit. These devices collectively manage a wide range of home automation functions, including lighting, locks, heating/cooling, leak detection, and security alarms. By mixing different device vendors and protocols, we ensure that the testbed captures the complexity of multi-platform smart homes found in the real world.

Table 1. Smart-home devices used in the evaluation.

Abbr.	Device Name	Protocol	Attributes
L1, L2, L3	Philips Hue light bulb	Wi-Fi	switch
S1, S2	THIRDREALITY Zigbee Switch	Zigbee	switch
P1	EIGHTREE Zigbee Smart Plug	Zigbee	switch, heater
P2	EIGHTREE Mini Smart Plug	Wi-Fi	switch, camera
M1	Aqara motion sensor	Zigbee	motion
M2	SmartThings motion sensor	Zigbee	motion
M3	Blue by ADT motion sensor	Z-Wave	motion
M4	Philips Hue motion sensor	Zigbee	motion
H	Amazon Ceramic Space Heater	–	heater
LK1	Kwikset Kevo Lock	Zigbee	lock
LK2	U-Bolt Pro Z-Wave Lock	Z-Wave	lock
SD	First Alert smoke detector	Wi-Fi	smoke
SR	Aeotec Siren 6	Wi-Fi	alarm
WL	SmartThings leak sensor	Zigbee	water
WV	Tuya Smart Valve Controller	Zigbee	water
CAM	Eufy indoor camera	Wi-Fi	motion, camera
SS	SimpliSafe security system	Wi-Fi	security system

Automation Rules A total of 20 automation rules govern various household scenarios, ranging from motion-triggered lighting to door locking/unlocking based on presence. Table 2 details the triggers, conditions, and actions of each rule, as well as the target device. We purposely include overlapping rules to provoke out-of-order events and potential contradictions when multiple platforms issue commands simultaneously (e.g., turning a lock on and off). Most rules run on three popular platforms: Amazon Alexa, Apple HomeKit, and Google Home. Several rules purposely overlap (e.g., R5 and R13 both control the same lock but trigger on distinct conditions), inducing potential concurrency conflicts. Locks, heaters, and high-stakes devices such as the water valve were carefully tested under IRB-approved guidelines to minimize resident inconvenience while allowing precise measurements of IoT-Handler's behavior. Some rules (R6, R9, R19, and R20) control water valves and leak sensors. In our setup, a Zigbee-based smart plug simulates a real water valve for safety reasons. Likewise, smoke- and presence-detection rules were tested under controlled conditions.

Table 2. Smart-home automation rules.

ID	Trigger Condition	Action	Platform(s)
R1	M1 motion (bedroom)	Turn on S1	Alexa
R2	M3 motion (living room) + luminance < 15 lux	Turn on L1, L2	HomeKit
R3	M2 motion (door) & M1, M3 inactive	Turn off L1, L2, S1, S2	HomeKit
R4	Time = 22:00 & LK locked	Turn on CAM	Google Home
R5	User arrives home	Unlock Zwavelock	HomeKit
R6	Water sensor triggered	Open water valve	Alexa
R7	Illuminance > 30 lux & M3 inactive	Turn off lights 1–3	HomeKit
R8	M3 motion	Turn on heater	Alexa
R9	Temperature $> 75\,^{\circ}$F	Turn off heater; open window	HomeKit
R10	Time = 10:00	Turn off CAM	Google Home
R11	M2 motion (door) & away mode	Turn on CAM	HomeKit
R12	SD detects smoke	Sound SR	Alexa
R13	M3 motion & home mode	Lock Zwavelock	HomeKit
R14	M4 motion & time $>$ 22:00	Lock ROOMLOCK; turn off heater	HomeKit
R15	Motion & home mode	Unlock Zwavelock2	HomeKit
R16	Door lock unlocked & heater off	Turn on heater	Alexa
R17	Room lock unlocked & heater on	Turn off heater	Alexa
R18	M2 inactive & away mode & 10:00–22:00	Turn off CAM	Google Home
R19	Leak sensor inactive	Turn off valve	Alexa
R20	Leak sensor active	Turn on valve	Alexa

5.2 Correctness and Preventing Conflicts

To systematically evaluate correctness under concurrency, we reproduced the message delay attack presented by Fu et al. [13] by selectively delaying IoT messages by 5–10 s at a programmable Wi-Fi router connected to IoT-Handler. We used eight representative conflict scenarios (C1–C8) drawn from the overlapping rules. Table 3 outlines these case studies, each involving a combination of two or three rules that can be triggered in close succession and conflict with each other. The table lists the rule IDs involved and briefly describes the nature of each conflict. These scenarios cover common categories of smart-home conflicts and race conditions: simultaneous sensor triggers targeting the same device, contradictory commands (one rule turning a device on while another turns it off), and cascading sequences of actions that overlap in time.

For each concurrency case C1–C8, we triggered the relevant rules in quick succession (either through real actions or scripted events) to observe the system's behavior with and without IoT-Handler. We found that **without IoT-Handler's mediation**, these scenarios often led to incorrect or inconsistent device states, as detailed below. Prior work has noted that such misordered IoT events and commands can leave devices in unsafe states, underscoring the importance of handling these conflicts.

In these and the other cases (C5–C8), IoT-Handler consistently maintained the correct execution order and state consistency. The root cause of each conflict was the same: physical-world events (motion, arrivals, and scheduled times) that can naturally occur almost simultaneously led to logically overlapping rules. IoT-Handler's design, which combines timestamp-ordered event queuing and context encapsulation, addresses this problem by isolating each rule's execution.

Table 3. Concurrency case studies (C1–C8) arising from overlapping automation rules. Each case involves multiple rules firing nearly simultaneously, producing conflicts that IoT-Handler must resolve.

Case	Rules Involved	Conflict Description
C1	R1, R3	Two motion sensors trigger overlapping actions on the same switch (S1), causing toggle conflicts.
C2	R5, R13	Presence detection and hallway motion both attempt to control the same door (ZL2), issuing potentially contradictory lock commands.
C3	R14, R15	Nighttime auto-lock of the front door (R14) vs. bedroom-door unlock on presence (R15) could cause lock oscillation.
C4	R14, R15, R16	Night routine issues three near-simultaneous actions (lock one door, unlock another, toggle heater), all overlapping in time.
C5	R2, R6	Light automation via motion (R2) overlaps with an environmental rule (R6) turning off the same lights, causing a race condition.
C6	R10, R11	"Away mode" routine turns off the camera (R10) while a motion event triggers the camera back on (R11), in conflict.
C7	R8, R14	Morning motion turns the heater on (R8) just after a night rule (R14) turns it off, leading to immediate reversal.
C8	R17, R18	Exit routine disables the thermostat (R17) while an entry routine enables heating (R18) at nearly the same time.

It freezes the affected device states for a few seconds when needed so that one rule's action cannot interfere with another's. In every concurrency test, the final device states under IoT-Handler matched the expected outcomes for the user's intentions, with no incorrect toggles or lost commands. By contrast, without the handler, we observed failures such as lights remaining off when two "on" commands raced, locks ending unlocked when two lock commands conflicted, or thermostats being re-enabled immediately after they were turned off. These findings reinforce that enforcing ordering at a fine-grained level is essential for reliability in multi-platform IoT environments, a challenge identified in prior research on misordered IoT events. IoT-Handler resolved all detected conflicts using its context-encapsulation strategy, with minimal impact on independent, non-conflicting operations.

5.3 Latency

We conducted our evaluation over multiple days. We performed both controlled experiments (in which we manually triggered events in specific sequences or introduced simulated network delays) and natural tests (in which the occupant followed a normal routine and we logged the system behavior).

Under non-conflicting conditions, a single rule experiences only minimal IoT-Handler overhead. The additional function calls from the handler add about 0.2–0.5 s to the baseline device round-trip time, aligning with previous studies of local

IoT system frameworks. For typical tasks such as turning on a light or turning off a plug, this overhead is negligible and does not affect the user experience. Moreover, the 0.2–0.5 s latencies remain within or below the standard 3–5 s cloud round-trip times for typical IoT devices.

Delay Time Trials A preliminary experiment introduced artificial delays in dispatching one command (C1) to observe IoT-Handler's success or failure in reconciling subsequent actions. Table 4 summarizes results for delay intervals ranging from −10 s to 60 s relative to a second event. We tested sequences involving Sensor 2, Sensor 3, and a switch, as well as sequences involving a multisensor, a TriSensor, and Z-Wave Lock 2. A successful outcome ("T") indicates that IoT-Handler properly synchronized the commands and prevented an inconsistent final state; a failure ("F") indicates that a contradictory or lost command occurred when mediation was bypassed.

Table 4. Command success (T) or failure (F) at different delay intervals.

Scenario (delay in s)	**-10**	**0.5**	**1**	**2**	**3**	**4**	**5**	**8**	**10**	**30**	**60**
2 sensors + switch	F	T	T	T	T	T	T	T	T	T	T
Multi6 + ZLock2	F	T	T	T	T	T	T	T	T	T	T
Multi6 + ZigbLock	F	F	T	T	T	T	T	T	T	T	T

When delays were opposite to the original intention or out of sequence (e.g., −10 s), the unsynchronized approach led to partial or dropped actions, producing "F" entries in the table. Once IoT-Handler enforced the correct chronological order (for any nontrivial positive delay), the final states consistently matched the user's intention.

In each of the above cases, the potential conflict arises because the physical triggers (motion, timers, or voice commands) can occur nearly simultaneously in real use. IoT-Handler's timestamp-based queuing (illustrated in Figure 4) ensures that rule actions are executed in a consistent and well-defined order. This preserves correctness even under adversarial timing or network delays: it eliminates race conditions and guarantees that each rule sees a coherent device state.

A second experiment recorded T_1–T_4 timestamps for overlapping rule sets. We compared direct device-to-platform paths (without IoT-Handler) with IoT-Handler-mediated execution (with a 5 s concurrency delay). Table 5 lists selected results, and Section 4 explains the timing sequence.

User feedback confirms that in time-critical tasks (e.g., quickly toggling lights), few conflicting triggers emerged, making IoT-Handler's overhead negligible. More often, concurrency problems appear with locks and heaters, for which mechanical operations lasting several seconds overshadow IoT-Handler's 5 s scheduling delay.

Table 5. Illustrative T_1–T_4 timing for overlapping rules with and without IoT-Handler (times in seconds).

Rule Pair(s)	Without IoT-Handler				With IoT-Handler (5 s freeze)			
	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4
R1 & R3	0.0000	0.7052	5.7053	5.7180	0.0000	0.7072	5.7084	5.7923
R5 & R13	0.0000	0.5331	5.5332	5.5403	0.0000	0.5433	10.3507	10.3762
R14 & R15	0.0000	0.5354	5.5354	5.5381	0.0000	0.5262	11.5347	11.5348
R14, R15, R16	0.0000	0.4461	5.4468	5.4502	0.0000	0.4525	11.4601	11.4623
R8 & R14	0.0000	0.5284	5.5289	5.5337	0.0000	0.5367	5.6325	5.6332
R14, R15, R17	0.0000	0.6123	10.6128	10.6219	0.0000	0.6151	10.8192	10.8274
R1 & R2	0.0000	0.6231	5.6232	5.6345	0.0000	0.6255	5.6733	5.6948
R1 & R3 (run 2)	0.0000	0.5124	5.5125	5.5198	0.0000	0.5147	5.7126	5.8201
R11 & R18	0.0000	0.4810	5.4819	5.4893	0.0000	0.4832	5.5812	5.6894
R4 & R18	0.0000	0.4510	5.4512	5.4587	0.0000	0.4533	5.4514	5.5639

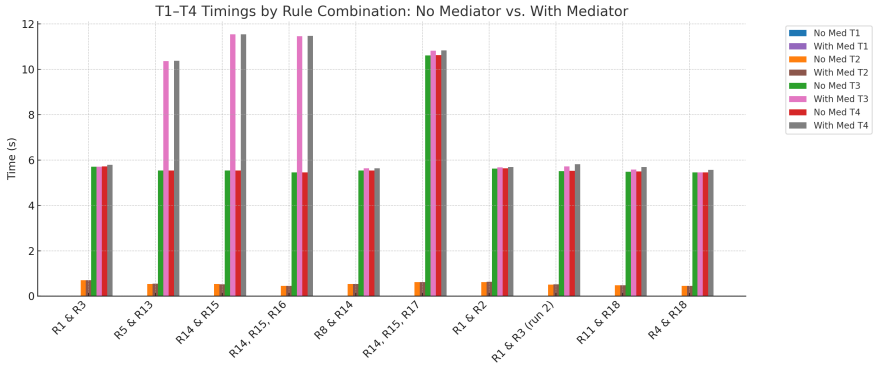


Fig. 7. Bar chart of T_1–T_4 timing for overlapping rules with and without IoT-Handler.

6 Related Work

6.1 IoT Message Delay

Fu et al. [13] and Chi et al. [5] introduced IoT message delay attacks in single- and cross-platform scenarios, respectively. They demonstrated that by selectively delaying messages from specific devices, attackers can manipulate the arrival order of IoT messages, resulting in erroneous or disabled automation-rule executions. Goksel et al. [15] discuss the potential consequences of misordered events and commands in IoT automation systems without considering adversaries and quantitatively analyze the likelihood of these scenarios. Their results show a high probability that messages from different IoT devices become misordered during transmission if they occur within a very short time span.

6.2 IoT Delay Defense

Several defenses have been proposed to address delay-induced IoT inconsistencies. MP-Mediator [28] is the closest to our approach: it acts as an intermediate monitor to detect and handle stealthy delay attacks. MP-Mediator interposes on IoT message flows using virtual devices and rules as hooks and deploys VPN-based proxies to capture delayed traffic. By monitoring inter-event timing, it can quickly identify abnormal latencies with high accuracy (over 96% precision and 100% recall) and then mitigate their effects to maintain correct operation. However, MP-Mediator's strategy is fundamentally detection-and-reaction: it requires integration with existing cloud platforms (e.g., SmartThings [23] and IFTTT [17]) to observe delays and does not prevent the initial desynchronization from occurring. TAPFixer [30] models trigger–action–platform (TAP) rules as latency-aware automata and uses model checking to patch *action-overriding* races. It cannot, however, address single-rule context inconsistencies or pure postponement attacks. Broader safety frameworks such as IoTGuard [4], IoTSafe [10], RuLedger [11], and HAWatcher [14] enforce logic-level policies but assume correct event ordering.

In contrast, our proposed IoT-Handler proactively preserves temporal consistency by design, isolating each rule's context and sequencing events in real time before conflicts arise. This ensures that all platforms receive events in a coherent order, eliminating the window of vulnerability that detection-based solutions must detect.

6.3 Security in IoT Automation

Several existing studies have addressed security challenges related to automation-rule execution and communication channels within IoT environments. Sengupta et al. introduce Transactuations [24], which investigates inconsistencies between physical states and IoT automation applications. Such inconsistencies may result from failures during application execution or incorrect concurrency management. To mitigate these inconsistencies, the authors propose a failure-aware programming model that enables developers to create robust and fail-safe automation applications. Similarly, Newcomb et al. propose IoTA [19], a formal modeling framework tailored for IoT automation. This framework provides structured methods to accurately represent and reason about interactions, dependencies, and workflows, ensuring that automation-rule executions adhere to specified safety properties. In VoiceGuard [29], Xu et al. propose a method that blocks unauthorized voice commands by delaying voice-command network packets and performing out-of-band user-presence verification. Fu et al. present IoTSentry [12], which uses generic home-security cameras to monitor state changes in smart-home IoT devices. The observed states serve as ground-truth information for detecting spoofed IoT events. In PFirewall [7], Chi et al. design a semantics-aware network packet filter that dynamically blocks IoT events from being reported to cloud servers, preserving user privacy without affecting home-automation functionality.

Norris et al. [20] present an IoT application-development framework that inherently supports automatic handling of device faults. Their approach includes functions such as redundant device activation, state checkpointing, and rollback mechanisms, thereby enhancing IoT-system resilience against failures. IoTGuard [4] and IoTMon [9] explore potentially hazardous interactions among IoT automation rules and applications, highlighting risks of conflicts or unintended chains of interactions within the physical environment.

Building further upon IoTMon, IoTSafe [10] introduces a runtime method for discovering covert physical interaction channels between applications, offering a refined and precise identification of these hidden communication pathways. Addressing the security challenges identified in these studies, Ozmen et al. propose IoTSEER [21], which converts IoT device messages into composable physical execution models. This method enables the verification of automation applications against predefined safety and security policies, incorporating an awareness of potential physical interactions prior to their deployment.

7 Conclusion

In this paper, we presented IoT-Handler, a practical and effective solution for preventing erroneous executions of automation rules in multi-platform smart-home environments. Through systematic analysis, we identified that unsynchronized event timing across platforms creates exploitable vulnerabilities that can lead to unsafe device states. IoT-Handler addresses these challenges by introducing execution-context encapsulation, timestamp-based arbitration, and selective event queuing, which isolate each rule's context while maintaining overall system responsiveness. In particular, our selective event-queuing mechanism efficiently handles device dependencies by allowing parallel processing of unrelated IoT events and guaranteeing strictly ordered execution for dependent actions. Our real-world evaluation demonstrated that IoT-Handler achieves 100% correctness in preventing race conditions and conflicting rule outcomes, with only a small performance overhead ($\approx$ 0.2–0.5 s per event). It also reduces worst-case automation delays from minutes (in unprotected systems) to seconds. Importantly, IoT-Handler requires no changes to existing IoT devices or cloud platforms, making it a readily deployable defense against stealthy message-delay attacks and benign network delays in multi-platform IoT ecosystems that millions of people use daily.

Acknowledgments This work was supported in part by the United States National Science Foundation (NSF) under grants CNS-2204785, CNS-2205868, and 2409212.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

A Ethical Considerations and Mitigation

This study was conducted under rigorous research ethics protocols. The deployment and data collection were reviewed and approved by our Institutional Review Board (IRB), especially for experiments involving safety-critical devices (e.g., smart locks and heaters). All participants (in this case, the single resident and study operators) gave informed consent and were free to withdraw at any time. Consistent with standard practice, we treated the wired smart-home testbed as a controlled environment. No vulnerable populations were involved, and the deployment was monitored at all times by the research team to allow immediate intervention if needed.

All recorded data were handled with strict privacy and security safeguards. We collected only the minimum information necessary (no audio/video or sensitive PII) and did not record any personally identifying details beyond the fact of occupancy. Sensor and interaction logs were stored on secure, access-controlled servers and were de-identified and aggregated before analysis. In accordance with U.S. human-subjects regulations (45 CFR 46.111(a)(7)), we ensured "adequate provisions to protect the privacy of subjects and to maintain confidentiality of data." Specifically, raw logs were encrypted at rest, identifiers were stripped or replaced with pseudonyms, and only authorized project personnel could access the data. All results (e.g., system-latency and correctness metrics) are reported in aggregate form to prevent inference of individual behavior.

During testing, we maintained a supervised laboratory setting with real-time monitoring and manual overrides. We implemented strong data security (encryption, limited access, and de-identification), adhered to IRB-mandated procedures for safety-critical devices, and emphasized that IoT-Handler's purpose is defensive: closing the gaps that could otherwise be exploited. These measures collectively minimize risk to participants and future users.

B Real-World Examples of IoT Automation Tests

1. Case C1 (R1 & R3): Here, two motion sensors (bedroom and hallway) nearly simultaneously trigger the entryway smart switch (S1). In practice, this could occur when a person moves from one room to another, causing two automation rules to fire at almost the same time. Without IoT-Handler, the switch (which controls a lamp) would receive two concurrent "on/off" toggle commands, likely resulting in only a partial execution (e.g., the lamp might flicker or end up in the wrong state). IoT-Handler prevents this race condition by intercepting the triggers and ordering the two commands by their timestamps. It ensures that the action from the first motion event completes (S1 is turned fully on) before the second command executes, thereby avoiding a conflicting toggle and leaving the lamp in a consistent state. Case C2 is similar to C1.
2. Case C2 (R5 & R13): In this scenario, a presence-based rule (R5, sensing the resident's arrival) and a motion-triggered rule (R13, sensing movement

in the hallway) both attempt to lock the back door (device ZL2) around the same moment. We observed that without mediation, their lock commands (the T_3 messages from the cloud) often collided, effectively sending two "lock door" instructions to the smart lock within milliseconds of each other. This can confuse the lock (one command may interrupt the other, possibly leaving the door unlocked or in an error state). IoT-Handler detects the conflict and introduces a brief freeze before executing the second command. Concretely, when R5 and R13 fire together, the handler immediately executes the first lock command but delays the second by about 5 s. This ensures that the back-door lock finishes the first locking action before the second command is applied, resulting in a correct final state (door locked) with no interference. The approximately 5 s freeze observed in this case is an intentional concurrency-control delay inserted by IoT-Handler, not network lag or an attack, but a deliberate wait to avoid a lock-state collision.

3. Case C3 (R14 & R15): This case involves two conflicting door lock actions at night. Rule R14 automatically locks the front door at bedtime, while R15 unlocks the bedroom door when motion is detected. These could be triggered within seconds of each other – for example, a resident walks inside at midnight, setting off the bedroom motion sensor just after the nightly lock schedule. Without coordination, the front door might lock at the same time the bedroom door unlocks, or worse, rapid re-locking/unlocking could occur if the platforms retry, creating an oscillation. With IoT-Handler, however, such conflicts are averted. The handler timestamps each trigger and sequences the two lock actions in chronological order. In one test run, the front door lock command (R14) was issued just moments before the bedroom unlock (R15); IoT-Handler ensured the front door's action took effect first. Our logs show that without the IoT-Handler, the two lock/unlock commands were initiated virtually simultaneously (both at around 5.53 s on the timeline), but under IoT-Handler they were spread out (e.g., one at 5.54 s and the next at 11.53 s). This enforced separation guarantees that each door ends in the intended state (front door locked, bedroom door unlocked), with no interference.
4. Case C4 (R14, R15, & R16): Case C4 extends the above to a triple-action conflict. Here, the nighttime routine (R14) triggers three different actions in overlapping fashion: locking the front door (as before), unlocking the second (bedroom) door, and turning off the space heater (S2) for the night. In our stress test, these three rules fired in quick succession (within the same second). Without the handler, all three device commands (two locks and one plug switch) would collide; e.g., at around 5.45–5.54 s, concurrent instructions would be sent to different devices, potentially causing contradictory outcomes or overloading the network. IoT-Handler's concurrency control serializes these events. The DeviceSynchronizer queues and staggers the actions: in the experiment, it delays the second and third actions so that the commands execute sequentially over an interval from about 11.46 s to 11.53 s (T_3 to T_4). Although this introduces a noticeable delay between the first and last actions (a 6-second span), it prevents unsafe combinations of events. The

front door locks, the bedroom door unlocks, and the heater turns off, each at a distinct time, ensuring that a "heater on" command (if any) arrives first and that the heater is then turned off. The handler thus avoids a possible race when multiple actuators are triggered.

References

1. Amazon.com, Inc.: Amazon Alexa. https://alexa.amazon.com/ (2025), accessed: August 6, 2025
2. Apple Inc.: Apple HomeKit - Home app. https://www.apple.com/home-app/ (2025), accessed: August 6, 2025
3. Bruschi, D., Ornaghi, A., Rosti, E.: S-ARP: a secure address resolution protocol. In: 19th Annual Computer Security Applications Conference (ACSAC). pp. 66–74. IEEE (2003). https://doi.org/10.1109/CSAC.2003.1254311
4. Celik, Z.B., Tan, G., McDaniel, P.D.: IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24–27, 2019. The Internet Society (2019). https://doi.org/10.14722/ndss.2019.23326
5. Chi, H., Fu, C., Zeng, Q., Du, X.: Delay wreaks havoc on your smart home: Delay-based automation interference attacks. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 285–302. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833620
6. Chi, H., Zeng, Q., Du, X.: Detecting and handling IoT interaction threats in Multi-Platform Multi-Control-Channel smart homes. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 1559–1576. USENIX Association, Anaheim, CA (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/chi
7. Chi, H., Zeng, Q., Du, X., Luo, L.: PFirewall: Semantics-aware customizable data flow control for smart home privacy protection. In: Network and Distributed System Security Symposium (NDSS) (2021). https://doi.org/10.14722/ndss.2021.24464
8. Chi, H., Zeng, Q., Du, X., Yu, J.: Cross-App interference threats in smart homes: Categorization, detection and handling. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 411–423. IEEE (2020). https://doi.org/10.1109/DSN48063.2020.00056
9. Ding, W., Hu, H.: On the safety of IoT device physical interaction control. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 832–846 (2018). https://doi.org/10.1145/3243734.3243865
10. Ding, W., Hu, H., Cheng, L.: IoTSafe: Enforcing safety and security policy with real IoT physical interaction discovery. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21–25, 2021. The Internet Society (2021). https://doi.org/10.14722/ndss.2021.23368
11. Fan, J., He, Y., Tang, B., Li, Q., Sandhu, R.: RuLedger: Ensuring execution integrity in trigger-action IoT platforms. In: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications. pp. 1–10 (2021). https://doi.org/10.1109/INFOCOM42981.2021.9488687
12. Fu, C., Du, X., Zeng, Q., Zhao, Z., Zuo, F., Di, J.: Seeing is believing: Extracting semantic information from video for verifying IoT events. In: Proceedings of the

17th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec). pp. 101–112. ACM (2024). https://doi.org/10.1145/3643833.3656124
13. Fu, C., Zeng, Q., Chi, H., Du, X., Valluru, S.L.: IoT phantom-delay attacks: Demystifying and exploiting IoT timeout behaviors. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 428–440. IEEE (2022). https://doi.org/10.1109/DSN53405.2022.00050
14. Fu, C., Zeng, Q., Du, X.: HAWatcher: Semantics-Aware anomaly detection for appified smart homes. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 4223–4240. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/fu-chenglong
15. Goksel, F., Ozmen, M.O., Reeves, M., Shivakumar, B., Celik, Z.B.: On the safety implications of misordered events and commands in IoT systems. In: 2021 IEEE Security and Privacy Workshops (SPW). pp. 235–241. IEEE (2021). https://doi.org/10.1109/SPW53761.2021.00038
16. Google LLC: Google Home. https://home.google.com/ (2025), accessed: August 6, 2025
17. IFTTT Inc.: IFTTT: If This Then That (2025), https://ifttt.com/, web-based service for creating chains of conditional statements. Accessed: August 6, 2025
18. Meng, Y., Zhang, W., Zhu, H., Shen, X.S.: Securing consumer IoT in the smart home: Architecture, challenges, and countermeasures. IEEE Wireless Communications **25**(6), 53–59 (2018). https://doi.org/10.1109/MWC.2017.1800100
19. Newcomb, J.L., Chandra, S., Jeannin, J.B., Schlesinger, C., Sridharan, M.: IoTA: a calculus for internet of things automation. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 119–133 (2017). https://doi.org/10.1145/3133850.3133860
20. Norris, M., Celik, B., Venkatesh, P., Zhao, S., McDaniel, P., Sivasubramaniam, A., Tan, G.: IoTRepair: Systematically addressing device faults in commodity IoT. In: 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI). pp. 142–148. IEEE (2020). https://doi.org/10.1109/IoTDI49375.2020.00021
21. Ozmen, M.O., Li, X., Chu, A., Celik, Z.B., Hoxha, B., Zhang, X.: Discovering IoT physical channel vulnerabilities. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2415–2428 (2022). https://doi.org/10.1145/3548606.3560644
22. Rohatgi, V., Goyal, S.: A detailed survey for detection and mitigation techniques against ARP spoofing. In: 2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC). pp. 352–356. IEEE (2020). https://doi.org/10.1109/I-SMAC49090.2020.9243604
23. Samsung Electronics: SmartThings (2025), https://www.samsung.com/us/smartthings/, smart home automation platform. Accessed: August 6, 2025
24. Sengupta, A., Leesatapornwongsa, T., Ardekani, M.S., Stuardo, C.A.: Transactuations: where transactions meet the physical world. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 91–106 (2019)
25. Statista: Smart home - statistics & facts (2024), https://www.statista.com/topics/2430/smart-homes/, accessed: August 6, 2025
26. Ur, B., McManus, E., Pak Yong Ho, M., Littman, M.L.: Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI conference on human factors in computing systems. pp. 803–812 (2014). https://doi.org/10.1145/2556288.2557420

27. XArp: Advanced ARP spoofing detection (2006), https://web.archive.org/web/20230724074341/https://xarp.net/, archived July 24, 2023; accessed: August 11, 2026
28. Xu, X., Fu, C., Du, X.: MP-Mediator: Detecting and handling the new stealthy delay attacks on IoT events and commands. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16–18, 2023. pp. 46–62. ACM (2023). https://doi.org/10.1145/3607199.3607225
29. Xu, X., Fu, C., Du, X., Ratazzi, E.P.: VoiceGuard: An effective and practical approach for detecting and blocking unauthorized voice commands to smart speakers. In: 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 582–596. IEEE (2023). https://doi.org/10.1109/DSN58367.2023.00060
30. Yu, Y., Xu, Y., Huang, K., Liu, J.: TAPFixer: Automatic detection and repair of home automation vulnerabilities based on negated-property reasoning. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 4945–4962. USENIX Association, Philadelphia, PA (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/yu-yinbo
31. Zhou, W., Jia, Y., Yao, Y., Zhu, L., Guan, L., Mao, Y., Liu, P., Zhang, Y.: Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1133–1150 (2019)