On Achieving Reliable and Efficient Precondition Execution Enforcement in Internet-of-Things

Qian Zhou and Fan Ye
Department of Electrical and Computer Engineering, Stony Brook University
Email: {qian.zhou, fan.ye}@stonybrook.edu

***Abstract*—In IoT it is common that before a command can execute on a smart object, certain preconditions (on possibly other objects) should be met first to ensure safety or efficiency. Existing work has realized automatic precondition execution: when a user issues a command, her device automatically finds out all the precondition commands, and executes them in the correct order. However, security issues have not been considered: it assumes that a user device honestly follows the order it has been told to send commands to objects, and objects trust users thus do not check whether the preconditions are indeed met. In this paper we propose two strategies to enforce precondition execution order: 1) Snowball relying on signed declarations from precondition objects; 2) Onion using disposable access tokens encrypted by a trustworthy server. Our extensive analysis and experiments on a 20-node testbed show that both strategies are secure and reliable. Snowball has higher availability while Onion is more efficient and responsive: Onion uses 1.6/2.1 s to access 20 one-hop/multi-hop objects, 62%/54% of Snowball's time.**

***Index Terms*—IoT, Security, Building Automation**

I. Introduction

In IoT there are common constraints that before a command can execute on a smart object (i.e. IoT device), certain states (called *preconditions*) on other objects should be satisfied for safety or efficiency. E.g., before a fire sprinkler sprays, the outlets within its spraying range should be powered down to prevent electric shocks; before an AC starts giving off cool air, the doors/windows should be closed to save energy. A command may have multiple preconditions, and each precondition may have its own preconditions, leading to multiple, recursive preconditions structured in a directed acyclic graph called *precondition execution graph* (PEG) [1].

Usually, users have to manually track and execute the commands that set those preconditions true (called *precondition commands*) one by one. Such laborious, error-prone operations daunt users and cause safety risks. Recently, solutions like APEX [1] are proposed for automatic precondition execution in IoT: a user obtains from a trustworthy backend server a PEG where the sink state (i.e. the vertex with no successor) corresponds to a user command, and predecessors are preconditions; when the user issues a command, her device (e.g., smartphone) automatically executes the precondition commands in the PEG order, and executes the user command at the end.

However, existing work assumes user/IoT devices are always trustworthy, and faithfully follow the rule: i) a user device honestly follows the PEG order to send commands to objects, which trust users thus do not check whether the preconditions are indeed set; ii) objects faithfully execute commands from users and return their states. In reality, besides common attacks targeting a single message's integrity/freshness, there are special attacks focusing on tampering with execution order among different commands. A user device or object can easily violate precondition constraints intentionally (e.g., malicious devices) or unintentionally (e.g., implementation bugs). Mechanisms reliably enforcing precondition execution are needed by both devices to verify that preconditions are indeed met, before they start subsequent execution.

In this paper we propose two strategies for such enforcement. 1) ***Snowball*** achieves the goal based on state declarations generated and signed by predecessor objects in the PEG; successor objects verify them and ensure their preconditions are satisfied. The more downstream an object is located in the PEG, the more precondition declarations it needs to verify, which resembles snowballing. 2) ***Onion*** uses disposable access tokens generated and encrypted by the backend: a token for accessing a successor object is encrypted; to use it, the user must access the predecessor objects first and obtain a secret from each of them to construct the decryption key. Tokens are decrypted and released gradually, like an onion peeled layer by layer. We claim our contributions as follows:

- We identify multiple attacks tampering with precondition execution order in IoT, and propose a signing-based solution called Snowball that allows objects to verify the states of their precondition objects.
- We also devise an encryption-based solution called Onion that uses encrypted, disposable tokens where a token for accessing an object cannot be decrypted before its precondition objects are accessed.
- We implement them and conduct extensive analysis and experiments on a 20-node testbed. Both achieve secure and reliable order enforcement. Snowball needs less access to the backend, thus is more robust upon backend unavailability; Onion has higher efficiency and lower latency—it takes 1.6/2.1 s to access 20 one-hop/multi-hop objects, 62%/54% of Snowball's time.

II. Models and Assumptions

Node Types. We consider the backend, subject devices and objects. As is widely used in practice, the backend is *not* a single server, but a hierarchy of servers run by the admin to manage registered subjects/objects; it realizes a chain of trust, and resists collapse under the load and a single point of failure. Subjects (i.e. users) use subject devices (e.g., smartphones) to

operate objects (i.e. IoT devices). We focus on security design above the network layer, and assume these two types form a *ground network*, where nodes in proximity are wirelessly connected and multi-hop routing [2], [3] is available.

A subject or object (denoted as X) must first register at the backend to join the system. This is a common requirement in real enterprise environments. X is issued with an ID, private key K_X^{pri}, public key certificate $CERT_X$ (signed by the admin so it cannot be forged/altered), and the admin's public key K_{adm}^{pub} (so X can verify the admin's signatures). Besides, X receives a symmetric key K_X^{sym} (possessed by X and the admin only). $CERT_X$ is publicized by X in the ground network, cached by nearby objects and can be queried [4].

Node Resources. Subject devices have good computing resources (e.g., Galaxy S8 has an octa-core CPU, 4GB RAM). Small objects (e.g., smoke detectors, occupancy sensors) are usually constrained, battery-powered sensors, while larger ones (e.g., door locks, ACs) are resource-rich (like Raspberry Pi), wall-powered, and with actuators. Nodes are roughly time synchronized (e.g., error within tens of seconds).

As pointed out in [1], IoT precondition execution usually involves *actuator* devices, of which the actuations prepare the physical (mechanical/electrical/thermal, etc.) environment for subsequent actuations. E.g., outlets must be turned off before sprinklers spray water, windows must be closed before ACs work. Unlike tiny sensor devices, actuator devices (outlets/sprinklers/windows/ACs) are mostly resource-rich and wall-powered. Thus in this paper we focus on objects which have sufficient computing resources and energy for public-key operations (e.g., signatures) at reasonable speed. The support for resource-poor objects is discussed in Section VIII.

State & Precondition. A *state* is a predicate on an object's variable, denoted as $\langle obj : var\ opr\ value \rangle$ where obj, var, opr are object identifier, variable name and operator (e.g., $=, >, \in$). As defined in [1]: if state $\mathcal{A}$ must be true before a command can execute to set state $\mathcal{B}$ true, then $\mathcal{A}$ is $\mathcal{B}$'s *precondition*. E.g., $\langle outlet_1 : status = \text{`}off\text{'} \rangle \Rightarrow \langle sprinker_2 : status = \text{`}spray\text{'} \rangle$ means $outlet_1$ should be turned off before $sprinkler_2$ sprays.

A. Background in Automatic Precondition Execution

Model: One Controller, Many Objects. Precondition execution using this model has three steps: 1) A controller (a subject device, or an object which is delegated as a controller) obtains from the backend a PEG—a directed acyclic graph where each vertex is a state, the sink state (i.e. the vertex with no successor) corresponds to a controller command, and predecessors are preconditions; 2) When issuing a command, the controller finds out the command's PEG and converts each precondition to a *precondition command*, getting a *combo* consisting of the controller command and all of its precondition commands; 3) The controller performs ***one round of combo execution*** by following the PEG order to execute those commands: a command is sent to its target object only if it gets ***ripe for execution*** (i.e., all of its precondition commands have been successfully executed). One round ends if the sink object finishes execution or this round is aborted. And repeating this process (no matter executing the same or a different combo) counts as another round.

E.g., a watchdog is delegated as a controller. When detecting smoke, it decides to make the fire sprinklers spray; by checking and following the PEG, it first safely shuts down the PCs powered by the outlets below the sprinklers, then turns off the outlets, and finally triggers the sprinklers.

Another model distributes the controller roles among the subject device and multiple objects. However, multiple controllers suffer from PEG synchronization overhead and failures. Besides, when multiple users attempt to execute combos on the same object, using a single controller for each combo can handle contention arbitration and subsequent abortion/recovery [1] much more easily.

Security. Existing work including APEX considers no security issues, and assumes that: i) all messages are authenticated, fresh; ii) controllers follow the give PEGs to operate objects, which trust controllers thus do not check whether their preconditions are indeed set; iii) objects honestly execute commands and return their states to controllers.

We choose to work on the one controller model due to the model's simplicity and efficiency. We will secure its execution order against malicious subject devices and malicious objects. Security model and analysis can be found in Section VI.

III. Design Goals

Execution Order Enforcement. Any order specified by a backend-issued PEG must be reliably enforced during command execution, despite malicious subject devices and objects. Especially, to know if a command is ripe, an object should be able to check whether *all* the preconditions of the command (corresponding to all of its predecessors in the PEG) are met.

Availability. Command execution on objects should involve access to the backend as little as possible, so execution is still available upon backend machine failures or connection losses, which happen in reality despite dedicated maintenance.

Efficiency. Command execution should be efficient in aspects of computation cost and message overhead, such that execution latency is short and user experience is positive.

Non-Goals. We focus on protecting the order among different commands, while securing each single command (e.g., integrity, freshness) has been significantly studied and we leverage existing work [5]. We do not consider execution order confidential so secrecy is not a goal. Also, resistance to DoS attacks and physical level jamming is out of the scope.

IV. Strategy A: Snowball

Snowball achieves order enforcement based on signatures signed by predecessor objects in the PEG and verified by successor objects. In this strategy, each object returns to the controller a *signed* declaration of its state upon execution completion; when the controller sends a command to target object $\mathcal{O}$, it attaches the declarations from all of $\mathcal{O}$'s predecessor objects such that $\mathcal{O}$ can check if this command is ripe (yes if the preconditions are all set) before executing it. The

more downstream $\mathcal{O}$ is located in the PEG (i.e. closer to the sink), the more precondition declarations it needs to check—verification cost and message overhead snowball.

Our design is built on top of Heracles [5], a capability-based access control [6] system where a controller requests secure, unforgeable tokens called *tickets* from the backend, and uses them as proof of access rights during command execution. Thus once tickets are obtained, controllers can access objects even if the backend is unavailable. Heracles protects each single command's authenticity/integrity/freshness, and we bring in extra protection on execution order among commands.

A. Ticket Requesting

$\texttt{TKT}: [I_{TKT}, \mathcal{S}, \{State_x, State_y\}, T_{expiry}]SIG_{adm}$

Fig. 1: TKT in Snowball Strategy

Controller $\mathcal{S}$ requests from the backend a ticket (TKT) which is a token with a PEG embedded in. It is used to prove to objects that $\mathcal{S}$ is authorized to set any state in the PEG true.

Heracles Components. In Fig. 1, $I_{TKT}, \mathcal{S}, T_{expiry}$ denote TKT's ID, $\mathcal{S}$'s ID and TKT's expiry time. $[...]SIG_X$ is the content in brackets followed by a signature of it signed by entity X. Signed by the admin, TKTs cannot be forged/altered.

Snowball Components. Originally a Heracles TKT carries $\mathcal{S}$'s access rights for accessing one or more objects but no execution order is involved. Here we have a new component $\{State_x, State_y\}$, a set of precondition relationships where $State_x$ is $State_y$'s direct predecessor and multiple such relationships together constitute a PEG.

B. Command Execution

$\texttt{CMD}_{\mathcal{O}_j}: [I_{RND}, I_{CMD}, TKT, \{RES_{\mathcal{O}_i}\}_{\mathcal{O}_i \in \Lambda_j}, \mathcal{O}_j, F, T_{issue}]SIG_{\mathcal{S}}$
$\texttt{RES}_{\mathcal{O}_j}: [I_{RND}, I_{RES}, State_{\mathcal{O}_j}, T_{issue}]SIG_{\mathcal{O}_j}$

Fig. 2: CMD and RES in Snowball Strategy

Controller $\mathcal{S}$, if benign, follows the PEG order to send commands (CMD) to target objects. A CMD will be sent only after it gets ripe, i.e., its precondition objects are accessed and responses (RES) announcing success return. E.g., in Fig. 3 (a), initially the CMDs to object 1–4 are ripe thus sent; after the RESs return, the CMDs to object 5–8 get ripe and sent.

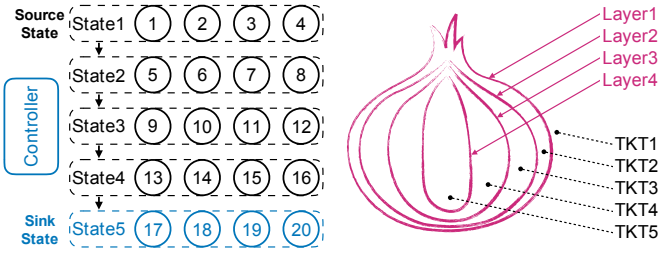
Heracles Components. In Fig. 2, $I_{CMD}, I_{RES}, \mathcal{O}_j, F$ and T_{issue} denote CMD's ID, RES's ID, target object's ID, the function to invoke, and the message's issue time. A CMD will be accepted by $\mathcal{O}_j$ only if it is: **1) authorized.** F aims to set a state authorized by TKT; **2) authenticated.** $SIG_{\mathcal{S}}$ is valid; **3) fresh.** Timestamp T_{issue} and nonce I_{CMD} together protect freshness: a CMD with a too old timestamp or a used nonce will be detected as replay and rejected.

Snowball Components. Malicious $\mathcal{S}$ does not follow the order, so we add new components I_{RND} (round ID) and $\{RES_{\mathcal{O}_i}\}_{\mathcal{O}_i \in \Lambda_j}$ for objects to perform execution order check: a CMD will be accepted only if it is also **4) ripe.** $\{RES_{\mathcal{O}_i}\}_{\mathcal{O}_i \in \Lambda_j}$ is a set containing the RESs from all of $\mathcal{O}_j$'s predecessor objects (denoted as Λ_j) in the PEG. $\mathcal{O}_j$ finds out its precondition objects Λ_j from TKT in the CMD, and regards the CMD ripe only if $\forall \mathcal{O}_i \in \Lambda_j$, a fresh RES (with fresh T_{issue} and I_{RES}) exists in the CMD's RES set and it carries the same round ID as the CMD, declares $State_i$ is set, and is signed properly by $\mathcal{O}_i$. E.g., in Fig. 3 (a), object 20 verifies the RESs of object 1–16 before executing the CMD.

Note that CMDs and RESs in one round of combo execution have the same I_{RND}, such that a CMD cannot put an RES of other rounds in its RES set as execution order proof. The round ID is recorded when the round ends, and subsequent messages with used round IDs will be rejected.

V. Strategy B: Onion

Onion achieves order enforcement based on encryption performed by the backend. The backend issues controller $\mathcal{S}$ beforehand with an onion-like token set whose outer layers are tickets (TKT) for accessing predecessor objects in the PEG and inner layers for successors. Initially, only the outermost layer TKTs (for accessing the PEG's source objects, i.e. those with no predecessor) are in plaintext while all the inner ones are *encrypted*. $\mathcal{S}$ uses plaintext TKTs to access the corresponding objects, which return the secrets needed to "peel" (decrypt) the Onion's current outermost layer and unseal the plaintext TKT of next layer. $\mathcal{S}$ repeatedly peels the Onion and accesses objects till the innermost TKT (for accessing the PEG's sink object) is obtained and consumed.



(a) Sample PEG (b) Sample Onion

Fig. 3: A sample PEG and the corresponding Onion

Since a TKT cannot be decrypted until the secrets from all of its predecessor objects are collected, the correct execution order is enforced. In Fig. 3: originally only TKT1 is in plaintext; $\mathcal{S}$ uses it on object 1–4 and each of them returns a secret; $\mathcal{S}$ uses the 4 secrets to compute a decryption key to peel Layer1, releases TKT2 and uses it on object 5–8; the process is repeated till TKT5 is released (this requires the secrets from object 1–16) and used on object 17–20.

A. Onion Requesting

$\texttt{TKT}_j: [I_{RND}, I_{TKT}, \mathcal{S}, \{State_x, State_y\}, State_j, T_{expiry}]SIG_{adm}$

Fig. 4: TKT in Onion Strategy

Controller $\mathcal{S}$ requests an Onion from the backend. Note that when using Snowball, the backend returns a PEG-level TKT which can be used to set any state in the PEG true, while here an Onion consists of multiple state-level TKTs (some are encrypted), with each being an access token for setting one state. E.g., in Fig. 4, TKT_j can only be used to set $State_j$ true. TKTs in one Onion have the same round ID, i.e. I_{RND}.

Onion Generator. In Algorithm 1, the backend first generates a TKT for each state and a secret ξ for each object of the input PEG (line 4–9). $\xi_{\mathcal{O}_j}$ is a hash-based message authentication code $\texttt{HMAC}(key, msg)$, key is the symmetric key $K^{sym}_{\mathcal{O}_j}$ issued to $\mathcal{O}_j$ at bootstrapping, possessed by only $\mathcal{O}_j$ and the backend. For each state $State_j$, if it is a source state (line 11–12), then TKT_j is directly put in the Onion. Otherwise (line 13–18) TKT_j is encrypted before being put. $(...)K^{enc}_j$ denotes the ciphertext of the content in parentheses encrypted using key K^{enc}_j, and K^{enc}_j is the **bitwise XOR** output of the secrets of $\mathcal{S}$ and all $\mathcal{O}_i \in \Lambda_j$, where Λ_j is $State_j$'s predecessor object set. E.g., in Fig. 3, TKT5 targeting State5 (object 17–20) is encrypted by a key generated using $\xi_{\mathcal{S}}$ and 16 object secrets $\xi_{\mathcal{O}_1}$–$\xi_{\mathcal{O}_{16}}$.

Algorithm 1 Onion Generator (run by the backend)

1: **function** ONIONGEN($\mathcal{S}, PEG$)
2: $Onion \leftarrow \varnothing$
3: $I_{RND} \leftarrow \texttt{Random}()$
4: **for all** $State_j \in PEG.vertices$ **do**
5: $TKT_j \leftarrow \texttt{TicketGen}(I_{RND}, \mathcal{S}, State_j)$
6: **for all** $\mathcal{O}_j$ of $State_j$ **do**
7: $\xi_{\mathcal{O}_j} \leftarrow \texttt{HMAC}(K^{sym}_{\mathcal{O}_j}, TKT_j.I_{RND} \| .I_{TKT})$
8: **end for**
9: **end for**
10: **for all** $State_j \in PEG.vertices$ **do**
11: **if** $State_j$ is a source state **then**
12: add TKT_j to $Onion$
13: **else**
14: $K^{enc}_j \leftarrow \texttt{HMAC}(K^{sym}_{\mathcal{S}}, I_{RND})$
15: **for all** $\mathcal{O}_i \in \Lambda_j$ **do**
16: $K^{enc}_j \leftarrow K^{enc}_j \oplus \xi_{\mathcal{O}_i}$
17: **end for**
18: add $(TKT_j)K^{enc}_j$ to $Onion$
19: **end if**
20: **end for**
21: **return** $Onion$
22: **end function**

B. Command Execution

$\texttt{CMD}_{\mathcal{O}_j}$: $[I_{CMD}, TKT_j, \mathcal{O}_j, F, T_{issue}]SIG_{\mathcal{S}}$
$\texttt{RES}_{\mathcal{O}_j}$: $[I_{RES}, State_{\mathcal{O}_j}, \xi_{\mathcal{O}_j}, T_{issue}]SIG_{\mathcal{O}_j}$

Fig. 5: CMD and RES in Onion Strategy

In Fig. 5, a command (CMD) using TKT_j as access right and execution order proof is sent to target $\mathcal{O}_j$, which returns secret $\xi_{\mathcal{O}_j}$ in its response (RES) upon execution success.

When receiving the RES from $\mathcal{O}_j$, $\mathcal{S}$ adds $\xi_{\mathcal{O}_j}$ to a set called $SecretSet$. Then as shown in Algorithm 2, it checks if any state in the Onion becomes ripe now: $State_{j+1}$ is ripe if $\forall \mathcal{O}_i \in \Lambda_{j+1}$ (predecessor), there is $\xi_{\mathcal{O}_i}$ in $SecretSet$. Then $\mathcal{S}$ computes the **bitwise XOR** output of $\xi_{\mathcal{S}}$ and those $\xi_{\mathcal{O}_i}$, gets K^{dec}_{j+1} which is identical to K^{enc}_{j+1}, and decrypts $(TKT_{j+1})K^{enc}_{j+1}$. This decryption "peels" the Onion, making its encrypted part smaller. Now $\mathcal{S}$ has TKT_{j+1} and can perform next access. In this way, $\mathcal{S}$ has to follow the PEG order to obtain TKTs and execute CMDs.

Algorithm 2 Onion Peeler (run by controller $\mathcal{S}$)

1: **function** ONIONPEEL($Onion, SecretSet$)
2: $NewTkt \leftarrow \varnothing$
3: get PEG, I_{RND} from $Onion$'s any plaintext TKT
4: **for all** $State_{j+1} \in PEG.vertices$ **do**
5: **if** $\forall \mathcal{O}_i \in \Lambda_{j+1}$, there is $\xi_{\mathcal{O}_i} \in SecretSet$ **then**
6: $K^{dec}_{j+1} \leftarrow \texttt{HMAC}(K^{sym}_{\mathcal{S}}, I_{RND})$
7: **for all** $\mathcal{O}_i \in \Lambda_{j+1}$ **do**
8: $K^{dec}_{j+1} \leftarrow K^{dec}_{j+1} \oplus \xi_{\mathcal{O}_i}$
9: **end for**
10: decrypt $(TKT_{j+1})K^{enc}_{j+1}$ with K^{dec}_{j+1}
11: add TKT_{j+1} to $NewTkt$
12: **end if**
13: **end for**
14: **return** $Onion$ and $NewTkt$
15: **end function**

Note that CMDs in one round of combo execution must use TKTs with the same I_{RND}. Each I_{TKT} is recorded till T_{expiry} is reached, for detecting TKT replay. Like Snowball, here I_{CMD}/I_{RES} are recorded at message reception, and I_{RND} is recorded when the round ends. Subsequent messages containing any used ID will be rejected.

VI. Security Analysis

Security Model. We assume the backend is trustworthy and well-protected. Its communication with subject devices or objects is secure. Subject devices/objects are reasonably well protected, e.g., by their OS. Also, we assume breaking the cryptographic algorithms (e.g., AES, ECDSA, in 128-bit strength) are computationally infeasible.

An attacker can capture, inject, modify or replay messages sent over the communication channel. Her goal is to compromise execution order, making unripe commands accepted. For expression simplicity, we denote the predecessor objects of $\mathcal{O}_j$ ($j \geq 2$) as $\mathcal{O}_1$–$\mathcal{O}_{j-1}$, with $\mathcal{O}_{j-1}$ being its direct predecessor. We show as below both strategies resist well attacks from a malicious controller unless the attacker compromises all the precondition objects; a rogue insider object causes limited harm and how to mitigate it is discussed.

A. Malicious Controller

A malicious controller can be a node impersonating benign controller $\mathcal{S}$ (denoted as $\mathcal{I}_{\mathcal{S}}$), or $\mathcal{S}$ which has gone rogue (denoted as $\mathcal{R}_{\mathcal{S}}$). $\mathcal{I}_{\mathcal{S}}$ poses as $\mathcal{S}$ to send unripe CMDs, which will be rejected for their invalid signatures, unless $\mathcal{I}_{\mathcal{S}}$ compromises $\mathcal{S}$'s private key. If $\mathcal{I}_{\mathcal{S}}$ does have the key, it becomes equivalent to $\mathcal{R}_{\mathcal{S}}$. $\mathcal{R}_{\mathcal{S}}$'s CMDs have correct signatures, and to make its unripe CMDs accepted, it just tampers with execution order proof (RES set in Snowball or TKT in Onion).

1) Proof Forgery. When using Snowball, $\mathcal{R}_{\mathcal{S}}$ may try to forge the PEG part in TKT to make target $\mathcal{O}_j$ believe that there is no precondition. It will fail since the integrity of TKT

is protected by the admin's signature. It may also forge the RESs from all of $\mathcal{O}_j$'s precondition objects, i.e. $\mathcal{O}_1$–$\mathcal{O}_{j-1}$, which needs the private keys of all of them.

When using Onion, $\mathcal{R}_\mathcal{S}$ may also forge TKT_j targeting $\mathcal{O}_j$, and fail due to the same reason. Or it may peel the Onion for getting TKT_j, but that needs the symmetric keys of $\mathcal{O}_1$–$\mathcal{O}_{j-1}$ for decryption key construction.

2) Proof Replay. When using Snowball, $\mathcal{R}_\mathcal{S}$ gets $RES_{\mathcal{O}_1}$–$RES_{\mathcal{O}_{j-1}}$ from the 1st round, and uses them to access $\mathcal{O}_j$ directly in subsequent rounds. However, Snowball has timestamps and nonces to prevent replay attacks: an object regards an RES as a valid execution order proof only if the RES has a new enough timestamp T_{issue} and its ID I_{RES} has not been received by the object before.

Similarly, when using Onion, $\mathcal{R}_\mathcal{S}$ only follows the PEG order once, gets all the TKTs and then replays them. It is also stopped by timestamps and nonces: an object regards a TKT as a valid proof only if the TKT is not expired and its ID I_{TKT} has not been received before. A received I_{TKT} can be removed from the tracking list after T_{expiry}.

Availability Comparison. Each Onion can be used only once because the TKTs within are disposable for anti-replay. $\mathcal{S}$ may estimate how many rounds it will execute, and request an appropriate number of Onions from the backend. Requesting too many is a waste, while too few leads to Onion exhaustion and the backend has to be accessed again. Upon backend unavailability, new Onions are unavailable, making objects unaccessible. A Snowball TKT can be used for arbitrary times before expiry, thus less access to the backend is needed, and higher availability in object access is achieved.

3) Proof Graft. $\mathcal{R}_\mathcal{S}$ may start the 1st round of combo execution and access $\mathcal{O}_1$–$\mathcal{O}_{j-1}$ in order, so it collects enough proofs to access $\mathcal{O}_j$; then it aborts this round, and uses another round (with a different PEG) to modify $\mathcal{O}_1$–$\mathcal{O}_{j-1}$ such that $\mathcal{O}_j$'s preconditions in the 1st round are no longer met; then it uses the 1st round proofs to access $\mathcal{O}_j$. Note that since the 1st round was aborted, $\mathcal{R}_\mathcal{S}$ has never used them on $\mathcal{O}_j$, so $\mathcal{O}_j$ cannot detect them as replay.

To resist this attack, in each round of combo execution: i) when using Snowball, RESs' I_{RND} must be identical to CMD's I_{RND} to get accepted by objects as valid execution order proof; ii) when using Onion, TKTs have the same round ID. When a round is completed, the sink object notifies all the predecessors; if an object receives an abortion signal, it notifies all the objects in the PEG. Either notification carries the current I_{RND}, and makes the receivers end the round, mark the round ID as used. Objects will discard subsequent messages with used round IDs, rejecting grafted proof.

B. Malicious Object

Similarly, the attacker can be a node impersonating benign object $\mathcal{O}_j$ (denoted as $\mathcal{I}_{\mathcal{O}_j}$), or rogue $\mathcal{O}_j$ (denoted as $\mathcal{R}_{\mathcal{O}_j}$). Unless $\mathcal{I}_{\mathcal{O}_j}$ compromises $\mathcal{O}_j$'s private key, its RESs will be discarded due to invalid signatures. As for $\mathcal{R}_{\mathcal{O}_j}$:

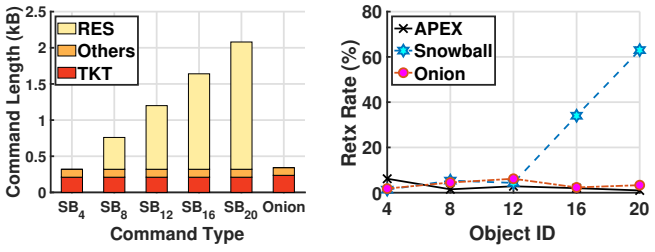
Proof Forgery. In both strategies, after receiving a CMD from $\mathcal{S}$, $\mathcal{R}_{\mathcal{O}_j}$ discards it and returns a dishonest RES announcing that it has set $State_j$ true. Then $\mathcal{S}$ moves on to send a CMD to $\mathcal{O}_{j+1}$, which actually is unripe. E.g., a window falsely claims that it is closed, then the AC is turned on when the room is unsealed. This attack does not compromise the entire PEG but only one edge, i.e. $State_j \Rightarrow State_{j+1}$. $\mathcal{S}$ may get input from objects besides the state's owner object for lie detection (e.g., an extra sensor can monitor the window's status), but a full solution is out of the scope.

VII. Performance Evaluation

Experiment Settings. We implement Snowball and Onion, and conduct real experiments on a 20-object testbed. To accurately evaluate the extra overhead our design brings into the existing system APEX [1], we follow its experiment settings: 1) Each object is emulated by a Raspberry Pi communicating via WiFi (the rationality of using resource-rich Pis is shown in Section II; the support for resource-poor objects is discussed in Section VIII); 2) The PEG in Fig. 3 (a) is tested; 3) Two network topologies are evaluated—one-hop situation where all the 20 objects are 1 hop away from controller $\mathcal{S}$, and multi-hop situation where object 1–4, 5–8, 9–12, 13–16, 17–20 are 1, 2, 3, 4, 5 hops away from $\mathcal{S}$ respectively. In reality $\mathcal{S}$ is a subject device, or an object delegated as a controller, so we use another Pi as an object controller first, and then show the case of a subject device controller. As for cryptographic algorithms, ECDSA (elliptic curve *secp256r1* [7], 128-bit strength) is used for signatures, AES (128-bit) for encryption.

A. Message Overhead

Without loss of generality, the number of objects in a PEG (n rows, m objects each row on average) is denoted as $N = mn$. We find Onion has much smaller message overhead than Snowball. An Onion command (CMD) has a constant number of components. A Snowball CMD starts with a similar length, but it increases with the number of predecessors, e.g., one targeting an object in the jth row carries $(j-1)m$ RESs. In our implementation, each signature has 64 B and an RES has 110 B, thus a Snowball CMD reaches up to kBs easily.



(a) Command length (b) Retransmission rate

Fig. 6: Message Overhead

Fig. 6 (a) shows the CMD lengths in our experiments. We choose object 4, 8, 12, 16, 20 in Fig. 3 (a) as representatives of different downstream degrees, which have 0, 4, 8, 12, 16 predecessors respectively. When using Snowball, the more downstream the target object is in the PEG, the longer the CMD is. E.g., the CMD for object 4 has 320 B while that for

object 20 has 2,080 B, because the latter additionally carries the RESs of 16 predecessor objects. In contrast, an Onion CMD is constantly 342 B long due to no object RES.

Fig. 6 (b) shows the CMD retransmission rates. We make a message resent if an ACK for it does not return within 200 ms. Such rates for APEX and Onion are always below 10% due to their short messages. Snowball's rate increases with message length, reaching 34% for a CMD targeting object 16 (1.6 kB long) and 63% for one targeting object 20 (2.1 kB).

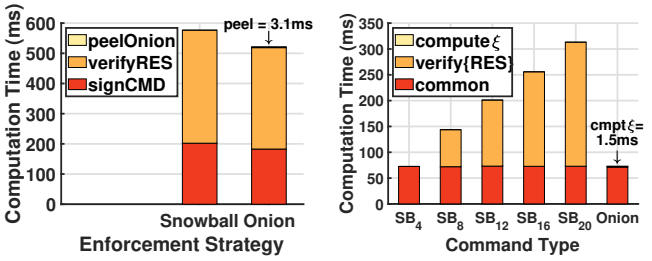
B. Cryptographic Operation Time Cost

We find Onion has similar computation cost to Snowball on the controller, but significantly smaller cost on the objects. Tab. I presents the number of public-key operations which dominates the computation time. On the controller, for both strategies, $\mathcal{S}$ needs to sign 1 CMD and verify 1 RES signature for each object, i.e. $2N$ operations in total. On the objects, the two strategies have 3 operations in common: each object verifies 1 CMD, 1 TKT, and signs 1 RES signature. When using Snowball, each of the m objects in the jth row ($2 \leq j \leq n$) additionally verifies the RESs of $(j-1)m$ predecessors. In total, Snowball needs $\sum_{j=2}^{n}(j-1)m^2 = 0.5m^2(n^2-n) = 0.5(N^2-mN)$ more public-key operations than Onion's $5N$.

TABLE I: The number of public-key operations

Strategy	Controller		Objects		Overall
	sign	verify	sign	verify	
Snowball	N	N	N	$0.5(N^2-mN)+2N$	$\mathcal{O}(N^2)$
Onion	N	N	N	$2N$	$\mathcal{O}(N)$

According to our experiments, on the controller Fig. 7 (a), for both strategies, $\mathcal{S}$ spends around 548 ms on cryptographic operations for 20 objects. Onion strategy additionally needs $\mathcal{S}$ to peel Onion layers, which is symmetric-key (AES) decryption and not compute intensive. The Onion in Fig. 3 needs to be peeled for 4 times, costing 3.1 ms only.



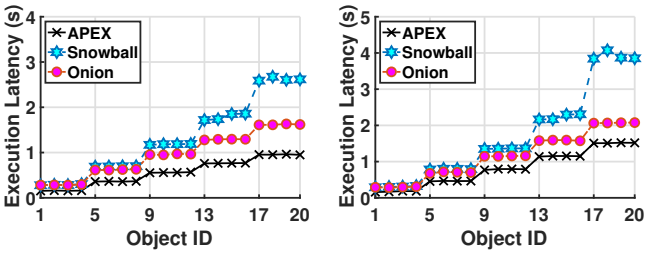
(a) Computation time, $\mathcal{S}$ (b) Computation time, $\mathcal{O}$

Fig. 7: Cryptographic Time

On an object Fig. 7 (b), the two strategies have 3 operations in common which cost 72.5 ms. Snowball needs objects to verify predecessor RESs: object 4, 8, 12, 16, 20 verifies 0, 4, 8, 12, 16 signatures respectively. The total computation time increases, from object 4's 72.5 ms to object 20's 313.2 ms. In contrast, Onion costs 74 ms in total for whichever object: though it needs each object to additionally compute a hash value ξ, that costs 1.5 ms only.

C. Execution Latency

We test the overall latency to access 20 objects in the PEG order, when APEX (with no security mechanism for order enforcement), Snowball, and Onion are used respectively, under one-hop and multi-hop situation. The ladder-shaped curves in Fig. 8 show that objects are accessed in the correct order (e.g., object 1–4 first, then 5–8). In one-hop situation Fig. 8 (a), APEX costs 1.0 s to access 20 objects; Snowball needs 2.6 s while Onion needs 1.6 s, about 62% of Snowball's time. We notice that the latency of Onion increases *linearly* as a more downstream object becomes the target; however, that of Snowball increases *quadratically*, because an object needs to verify the signatures from all of its predecessors.



(a) Latency, one-hop (b) Latency, multi-hop

Fig. 8: Execution Latency

In multi-hop situation Fig. 8 (b), similar phenomena are observed: APEX, Onion, Snowball cost 1.5 s, 2.1 s, 3.9 s respectively. Onion's advantage over Snowball in responsiveness becomes more obvious: it saves 1.8 s and costs 54% of Snowball's time. The latency difference becomes larger than one-hop because Snowball's long CMDs (those targeting object 17–20) have high retransmission rates (Fig. 6), and the more hops CMDs travel, the more time Onion saves.

Subject Device Controller. Controller $\mathcal{S}$ can also be a subject device (usually a smartphone). The computation time on a Samsung Galaxy S8 is 33 ms, shorter than Pi's 548 ms due to its better computing resource. So it can reduce both strategies's overall latencies by 0.5 s, but the time difference between Snowball and Onion does not change.

D. Summary

TABLE II: Comparison. ✓: advantageous item

Metric	Snowball	Onion
Security	Yes	Yes
Computation Cost	$\mathcal{O}(N^2)$	✓$\mathcal{O}(N)$
Message Overhead	up to kBs	✓10^2 bytes
Ticket Reusability	✓Yes	No

Onion has smaller message overhead, computation cost and execution latency, and may be preferred for its high efficiency and responsiveness. Snowball has better availability because its tickets (TKT) can be used repeatedly while Onions are disposable. Using them in combination may be recommended: $\mathcal{S}$ requests *multiple* disposable Onions plus *one* Snowball TKT; it uses Onions first for responsive object access, and uses the Snowball TKT as a provisional substitute after the Onions are exhausted and before new Onions are available.

VIII. Discussion

Support for Resource-Poor Objects. In reality resource-constrained, battery-powered objects may involve in precondition execution, and they can only run public-key algorithms slowly, occasionally. A delegation strategy [5] can make Snowball and Onion work on them: a constrained object establishes a symmetric key with a resource-rich object and delegates public-key operations to the rich one.

Another way is to replace compute intensive signatures in constrained objects' TKTs/CMDs/RESs with message authentication codes (MAC). Rich objects still use signatures. Specifically, the backend replaces SIG_{adm} with a MAC generated by $K_{\mathcal{O}_j}^{sym}$ to authenticate the TKT to constrained object $\mathcal{O}_j$. Similarly, controller $\mathcal{S}$ and $\mathcal{O}_j$ establish a symmetric key beforehand, and use it to authenticate their CMDs/RESs. Onion has less establishment overhead than Snowball. This is because Snowball requires objects to verify predecessor objects' RESs, thus symmetric keys should also be established between objects, besides between $\mathcal{S}$ and objects.

Resistance to Object Failures. We have shown in Section VI that our strategies resist well not only controller failures but also *object failures*. Though less analysis is on object failures than on controller ones, it is because this work focuses on securing the model of one controller and many objects. In such a "star topology" model, the controller is the pivot entity that coordinates overall execution order, thus can launch more attacks targeting execution order. Given a benign controller, a failed object can launch no attacks targeting execution order except lying about its state; this can at most violate one PEG edge, causing limited harm.

General Applicability. This work secures the order of actions among multiple devices. It is a generic problem beyond just precondition execution. Our solution is generally applicable to contexts where one or multiple devices are required to reliably follow a specified order to do a series of tasks. E.g., open a window to air the room for 10 minutes, and then close it; update firmware on all objects in a house, then reboot, and perform a suite of test operations on them.

IX. Related Work

Centralized Policy Enforcement. Many IoT solutions [8], [9], [10] use centralized execution for access control: subjects send commands to the cloud; the cloud checks if subjects have the access rights, and if yes, it forwards commands to target objects. CoMPES [11] is a cloud-based system with predefined policies of what commands to execute under what conditions; it receives condition information from objects, and sends commands back. CityGuard [12] is a centralized smart city safety watchdog enforcing safety policies.

Policy enforcement in these systems is trivial because objects only accept commands from the centralized entity which reliably carries out polices. But they are vulnerable to a single point of failure, have long execution latency, and may not be preferred especially in enterprise-scale IoT [5].

Distributed Policy Enforcement. There are also solutions using distributed execution [5], [13], [14], [15] which allow subjects to send commands directly to objects, without involving the centralized entity. However, what they do is ensure each individual command is valid (e.g., authenticated, authorized, fresh), and realizing order enforcement among multiple commands is essentially a different problem. Our strategies achieve distributed execution order enforcement.

X. Conclusion

In this paper, we describe the design, implementation and evaluation of two strategies for precondition execution enforcement in IoT: Snowball is signing-based while Onion is encryption-based. Our analysis and experiments on a 20-node testbed show that both strategies are secure and reliable. Snowball has higher availability while Onion is more efficient and responsive: Onion uses 1.6/2.1 s to access 20 one-hop/multi-hop objects, 62%/54% of Snowball's time.

Acknowledgment

This work is supported in part by the National Science Foundation under grant number 1652276.

References

[1] Q. Zhou and F. Ye, "Apex: automatic precondition execution with isolation and atomicity in internet-of-things," in *Proceedings of the International Conference on Internet of Things Design and Implementation*. ACM, 2019, pp. 25–36.

[2] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc on-demand distance vector (aodv) routing," Tech. Rep., 2003.

[3] A. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "Nlsr: named-data link state routing protocol," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. ACM, 2013, pp. 15–20.

[4] X. Song, Y. Huang, Q. Zhou, F. Ye, Y. Yang, and X. Li, "Content centric peer data sharing in pervasive edge computing environments," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 287–297.

[5] Q. Zhou, M. Elbadry, F. Ye, and Y. Yang, "Heracles: Scalable, fine-grained access control for internet-of-things in enterprise environments," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1772–1780.

[6] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[7] S. Blake-Wilson, B. Moeller, V. Gupta, C. Hawk, and N. Bolyard, "Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls)," 2006.

[8] Amazon, "AWS IoT Developer Guide," http://docs.aws.amazon.com/iot/latest/developerguide/iot-dg.pdf.

[9] SmartThings, "SmartThings Developer Documentation," https://media.readthedocs.org/pdf/smartthings/latest/smartthings.pdf.

[10] IBM, "Meet Watson: the platform for cognitive business," http://www.ibm.com/watson/ .

[11] J. Hall and R. Iqbal, "Compes: A command messaging service for iot policy enforcement in a heterogeneous network," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. ACM, 2017, pp. 37–43.

[12] M. Ma, S. M. Preum, and J. A. Stankovic, "Cityguard: A watchdog for safety-aware conflict detection in smart cities," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. ACM, 2017, pp. 259–270.

[13] S. Gusmeroli, S. Piccione, and D. Rotondi, "A capability-based security approach to manage access control in the internet of things," *Mathematical and Computer Modelling*, vol. 58, no. 5, pp. 1189–1205, 2013.

[14] W. Shang, Q. Ding, A. Marianantoni, J. Burke, and L. Zhang, "Securing building management systems using named data networking," *IEEE Network*, vol. 28, no. 3, pp. 50–56, 2014.

[15] N. Ye, Y. Zhu, R.-C. Wang, and Q.-m. Lin, "An efficient authentication and access control scheme for perception layer of internet of things," 2014.