# ReLES-OTA: A Reinforcement-Learning Enhanced Scalable Over-the-Air Update Approach for CAVs

Arpan Bhattacharjee
*Dept. of Computer and Information Sciences*
*University of Delaware*
Delaware, USA
arpan@udel.edu

Zheyuan Xu
*IEEE Member*
CA, USA
cx1014@uw.edu

Vikas Vyas
*Senior IEEE Member*
San Jose, CA, USA
talkwithvikas@gmail.com

Weisong Shi
*Dept. of Computer and Information Sciences*
*University of Delaware*
Delaware, USA
weisong@udel.edu

***Abstract*—Scalable over-the-air (OTA) updates for connected and autonomous vehicles (CAVs) present a significant challenge for many original vehicle equipment manufacturers (OEMs) due to the lack of scalability in their current software update solutions. We have proposed a novel reinforcement learning (RL) approach to tackle this issue. The problem-solving capability of deep reinforcement learning (RL) has been advanced in many fields, yet its potential for tackling over-the-air (OTA) update problems remains under-explored. As one of the backbones for enabling fast software iterations in various autonomous systems and robotic platforms, the OTA update problem remains challenging. This work primarily focuses on two key aspects of the OTA challenge: data payload size and memory overhead. By formulating the OTA problem and reducing it to a sequential codec construction problem and a Markov-Decision Process environment, we can apply deep reinforcement learning algorithms and generate approximately optimal OTA payload with small payload size and memory usage. Running experiments on real-world datasets, our proposed method performs better than open-source methods, effectively showcasing the potential of RL-augmented OTA update algorithms to reduce payload size and minimize runtime memory overhead. Finally, this work lays a robust foundation for applying deep RL in automotive OTA update systems, with potential implications for broader autonomous systems and robotic platforms.**

***Index Terms*—OTA Update, Deep Reinforcement Learning, Markov-Decision Process, Connected and Autonomous Vehicles (CAVs)**

## I. Introduction

Modern Connected and Autonomous Vehicles (CAVs) continuously evolve through software enhancements to their control systems, infotainment units, sensor fusion algorithms, and advanced driver assistance functions [1]. As traditional update methods - which require physical access or downtime in a service station - are no longer scalable [2], OTA Software updates are critical to patching security vulnerabilities [3], introducing new features (e.g., improved autonomy levels), and refining perception, planning, and control modules [4]. OTA updates enable remote management, streamline logistics, and ensure all fleets remain operationally cutting-edge [2].

As software systems become increasingly complex and security critical, frequent updates are necessary to patch vulnerabilities, introduce new features, fix bugs, or adapt system performance to changing operational requirements [5].

Despite their importance, OTA updates pose several technical challenges. Firstly, many devices operate in bandwidth-constrained environments—such as remote IoT sensors connected via low-power wide-area networks (LPWANs)—where transmitting large Software images is costly and can induce prolonged downtime [6]. Secondly, the growing complexity of modern software exacerbates these issues: larger update files require more bandwidth and time, potentially consuming substantial energy and risking unacceptable latency for time-critical systems [4]. Moreover, devices in the field may have limited memory and computational capabilities, constraining the complexity and overhead of update strategies [7]. Thus, a key engineering challenge lies in developing OTA update mechanisms that can dynamically optimize data transfer, minimize resource usage, and ensure the timely completion of the update process.

### *A. Motivation*

CAVs operate within bandwidth-constrained and latency-sensitive vehicular networks. Cellular connectivity, roadside units, and ad-hoc vehicular-to-vehicular links often impose strict throughput and reliability constraints [8]. Software packages can be large and complex, leading to significant transmission costs and potential delays that degrade the end-user experience or, in the worst cases, operational safety. To maximize efficiency, CAV engineers require adaptive OTA strategies that can dynamically select optimal update actions—like partial deltas, block-level modifications, or intelligent backup policies—tailored to the current software state and network conditions [9].

Recent progress in Reinforcement Learning (RL) provides a promising route. RL agents can learn policies that minimize patch sizes, bandwidth usage, and update times.

### B. *Research Gap*

Existing OTA optimization schemes in automotive contexts often rely on fixed heuristic strategies or delta-compression algorithms without adaptation [10]. To the best of our knowledge, there are literally no approaches that harness RL to improve policies continuously based on observed performance. This gap leaves untapped opportunities for more intelligent, context-aware OTA update methods in the CAV domain, where safety, compliance, and seamless user experience demand advanced adaptability. The current challenges facing the integration of RL with OTA update strategies include the following:

*1) **Lack of Mathematical Modeling**:* RL methods work best in the Markov-decision process (MDP) or partially observable Markov-decision process (POMDP), in which the problem has explicit states, actions, transitions between different states, and rewards [11]. OTA update strategies, on the other hand, usually aim to create the optimal representation of the update process, which lacks intermediate states and thus creates difficulty for the integration of RL algorithms.

*2) **Output Correctness**:* It is of vital importance to ensure the correctness and integrity of the OTA update algorithm, and it introduces additional mathematical modeling challenges. Usually, a task's correctness or completion is modeled as a final sparse reward at the end of the training episode, which is difficult for the neural network due to the sheer number of possible outcomes.

### C. *Contributions*

We propose a novel RL-based OTA framework for full-system OTA updates. While many existing OTA algorithms focus on updating specific modules, such as neural network models or maps, they generally require the modules to be decoupled from the rest of the system and require file-level read-write access rights. Often known as more secure and less-constrained solutions, full-system updates treat the entire operating system image as multiple data blocks and are agnostic to the file system and content.

We introduce an MDP-based formulation of OTA updates for CAV software. In order to do that, we focused on two aspects of OTA algorithms, updated patch size and runtime memory overhead. In addition to that, adaptive policies that consider system constraints and network conditions are enabled. Furthermore, we utilized the open-source OpenAI Gym framework to design a custom Gym environment and a sample PPO training pipeline, including a stable baseline RL agent and code adaptations to handle CAV-specific full-system OTA updates.

The rest of this paper is organized as follows: Sec. II reviews the background and related works about OTA updates. Sec. III presents the mathematical modeling behind our OTA update approach. Sec. IV describes the step-by-step procedure of our proposed block-wise OTA update strategy in detail. Extensive experimental results are shown in Sec. V and finally we conclude the entire paper discussion in Sec. VI.

## II. Related Work

### A. *OTA Update Strategies in Automotive and IoT Systems*

OTA updates are a critical enabler of continuous software improvement in automotive systems, IoT networks, and mobile devices. Traditional methods for OTA updates focus on minimizing bandwidth usage and reducing update patch size. Key strategies include:

*1) **Delta-Based Compression Tools**:* Tools like BSDiff [12], Xdelta [13], and Rsync [14] generate compact patches by encoding only the differences between two versions of a file. These methods achieve significant bandwidth savings but often struggle with high computational overhead and might suffer in certain scenarios due to their hard-coded heuristics.

*2) **Chunking and Content-Aware Encoding**:* Chunking strategies divide large software files into smaller blocks for efficient transmission. Simple heuristics and context-aware encodings can be used to greatly reduce the update patch size and reduce runtime memory cost for the update [15]. For example, identical blocks can be represented by very few special bytes in the update patch and when parsed by the target system, would be converted into a copy operation from the desired block.

*3) **IoT-Specific Challenges**:* IoT networks face unique constraints, including low-power devices and limited memory. Lightweight communication protocols such as LWM2M [16] and CoAP [17] optimize resource usage but do not extend to CAV-level complexity, where the need for local hardware resource usage compounds bandwidth constraints.

While these methods excel in static or low-complexity environments, they lack adaptivity to the highly dynamic and latency-sensitive nature of CAV networks. Thus, the need for an intelligent, learning-based OTA strategy is evident.

### B. *Deep Reinforcement Learning in Network and Edge Intelligence*

Deep Reinforcement Learning (RL) has demonstrated remarkable success in solving complex decision-making problems across various domains, from Atari games [18] to resource allocation [19]. Its application in network and edge intelligence is particularly relevant to the challenges of OTA updates.

*1) **Resource Allocation and Load Balancing**:* RL-based techniques have been employed to allocate resources in wireless networks, optimize task offloading in edge computing, and manage network congestion. For example, Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO) [20] have been used to optimize throughput and reduce latency [21] in multi-user systems.

*2) **Adaptive Policies for Changing Environments**:* Unlike static heuristics, RL enables agents to learn adaptive policies that respond to dynamic conditions such as fluctuating bandwidth, latency, and device heterogeneity [22]. This adaptability is crucial for CAV OTA updates, where network conditions and software complexity vary significantly across scenarios.

*3) **Applications in Automotive Systems**:* In the automotive domain, RL has been applied to tasks such as:

- **Vehicular Routing:** Optimizing routes in connected vehicle networks to reduce congestion and improve travel times [23].
- **Driver Assistance Systems:** Enhancing adaptive cruise control and collision avoidance using RL to optimize system response [24].
- **Power Management:** Using RL to optimize battery usage and minimize energy consumption in electric vehicles [25].

Extending RL to OTA updates represents a natural progression. By framing OTA updates as a sequential decision-making problem, RL agents can intelligently choose actions that minimize patch sizes and resource overhead.

### *C. **Existing Scientific Approaches to OTA Update***

In this section, we will present a comparative study of the scientific works made in the literature on over-the-air (OTA) updates for CAVs, focusing on resolving scalability, security, and efficiency issues. Researchers [26] proposed a linear programming-based approach to decrease handovers during updates from fog nodes; however, this approach carries the risk of overloading these nodes and impairing the performance of delay-sensitive applications. Techniques like genetic algorithms and tabular search have been used to investigate optimization-based approaches, such as time and machine-dependent scheduling [27]. Genetic algorithms have demonstrated superior performance in reducing update completion times, making them a promising solution.

Incremental update methods like MoRE [28] drastically cut data transfer sizes and eliminate the need for extra memory at sensor nodes by concentrating on sending only the variations between firmware versions. Hardware firewalls and secure controller area network protocols have been developed to protect electronic control units (ECUs) from attacks [29]. Another strong option that has surfaced is blockchain frameworks, which guarantee the integrity of OTA firmware upgrades [30][31]. Furthermore, software-defined transmission control protocols make dynamic scheduling possible, solving issues like dynamic topology and unstable links and meeting the particular needs of vehicle networks.

Despite continued improvements, existing methods often use centralized fog nodes to disperse updates, leading to limited scalability, single points of failure, and network congestion [32]. Furthermore, many methods are not thoroughly evaluated, which ignores the growing demands of the quickly growing connected car market. The necessity for scalable and secure solutions is highlighted by introducing new device-to-device communication systems based on pivot nodes as decentralized solutions. These solutions are crucial to adapt to the evolving software-defined vehicle landscape, where the focus is on overcoming limitations, dividing the workload, and strengthening system resilience [33].

Table I provides a comprehensive summary of the notations used in this work to assist readers in navigating the mathematical framework. It includes variables, functions, and parameters relevant to the Markov Decision Process (MDP), reward function, and policy optimization, ensuring clarity and ease of understanding.

TABLE I: Table of Notations

| **Notation** | **Description** |
|---|---|
| $O$ | Old software, represented as a set of blocks $O = \{o_1, o_2, \ldots, o_n\}$ |
| $N$ | New software, represented as a set of blocks $N = \{n_1, n_2, \ldots, n_m\}$ |
| $o_i$ | $i$-th block in the old software |
| $n_j$ | $j$-th block in the new software |
| $H(o_i)$ | MD5 hash of the $i$-th block in the old software |
| $\mathcal{S}$ | State space of the Markov Decision Process (MDP) |
| $b_t$ | Index of the current block being processed at timestep $t$ |
| mask | Binary vector indicating the processing status of each block (1 for unprocessed, 0 for processed) |
| $c_t$ | Cumulative encoding cost up to timestep $t$ |
| $m_t$ | Memory usage up to timestep $t$ |
| $\Delta(o_i, n_j)$ | Difference metric (e.g., hash difference or delta size) between $o_i$ and $n_j$ |
| $\delta(o_i, n_j)$ | Delta patch generated between $o_i$ and $n_j$, computed as: $bsdiff(o_i, n_j)$ |
| $s_t$ | State of the environment at timestep $t$. |
| $a_t$ | Action chosen by the RL agent at timestep $t$ |
| $\pi_\theta(a_t\|s_t)$ | Policy function (actor-network), representing the probability of choosing $a_t$ in state $s_t$ |
| $V_\phi(s_t)$ | Value function (critic network), estimating the expected cumulative reward from state $s_t$ |
| $R(s_t, a_t)$ | Reward obtained for taking action $a_t$ in state $s_t$ |
| $\gamma$ | Discount factor for future rewards ($0 \leq \gamma \leq 1$) |
| $\lambda$ | Weighting factor for Generalized Advantage Estimation (GAE) |
| $\alpha, \beta$ | Weighting factors for encoding cost and memory cost in the reward function |
| $C_{\max}$ | Normalization constant in the reward function |
| $M$ | Modify action, which updates the block using direct copy (MC) or delta compression (MD) |
| $MB$ | Modify + Backup action, which updates the block and creates a backup for rollback |
| $L^{\text{CLIP}}(\theta)$ | Clipped surrogate objective for PPO optimization |
| $r_t(\theta)$ | Probability ratio: $r_t(\theta) = \frac{\pi_\theta(a_t\|s_t)}{\pi_{\theta_{\text{old}}}(a_t\|s_t)}$ |
| $\hat{A}_t$ | Generalized Advantage Estimate (GAE) at timestep $t$ |
| $\delta_t$ | Temporal Difference (TD) error: $R(s_t, a_t) + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ |

## III. Mathematical Modeling

### *A. **Problem Setting***

Modern CAVs contain multiple Electronic Control Units (ECUs), each responsible for functions ranging from sensor fusion to infotainment. Manufacturers rely on Over-the-Air (OTA) software updates to maintain and enhance these systems remotely without human intervention, as illustrated in Fig. 1, in which the backend server generates update patch, which is delivered through the internet or radio communication infrastructure to the target CAV system. We assume ample computation and memory resources exist in the backend server, while the resources are relatively constrained on the target system.

Let the currently installed (old) software of a given CAV be represented as:

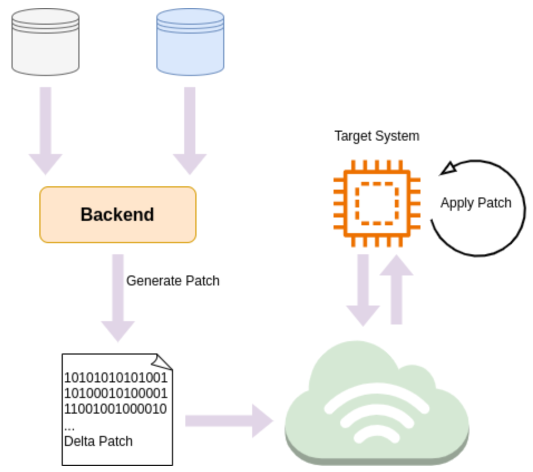


Fig. 1: OTA update workflow comprises a backend system that generates the update patch and a runtime system that receives the patch over the CAV connectivity infrastructure and applies the patch.

$$O = (o_1, o_2, \ldots, o_n)$$

and the target (new) software be:

$$N = (n_1, n_2, \ldots, n_m)$$

Each $o_i$ or $n_i$ is a binary block that stores code or configuration data. The number of blocks, $n$, may vary based on ECU memory architecture and software modularity. Typically, for large CAV software images, $n$ can be in the hundreds or thousands.

The objective is to transform $O$ into $N$ using a series of actions that minimize transmitted data, memory consumption, and update latency—while also preserving system integrity (e.g., enabling rollback if needed). These objectives are especially critical in CAVs, where software updates must not compromise real-time safety functions or disrupt critical subsystems.

## *B. Action Set*

To facilitate an intelligent, block-level update policy, we define a small yet expressive set of actions:

- **M - Modify Block:** Directly transform the current block $o_i$ into the target block $n_i$. This might involve:
    1) **Copy Update:** If a known reference block $o_j$ (in the old data blocks) is identical to $n_i$, we directly copy block $o_j$ and overwrite $o_i$ as illustrated in Fig. 2. This would lead to an encoding cost of 2 characters (MC) + length of the index of the block to be copied.
    2) **Delta Update:** If no identical block is found, we perform delta update on the "closest" block $o_j$ against $n_i$ and overwrite $o_i$ with the result, given whichever $o_j$ generates smallest delta patch against $n_i$ shown in the right part of Fig. 2. This would lead to an encoding cost of 2 characters (MD) + length of the index of the block to be delta-ed against + length of the delta patch generated.
- **MB - Modify and Backup Block:** Same as "Modify", but we also create a backup copy of the original block $o_i$ before overwriting it. This allows for a possible future copy update (MC) operation in case the backup block $o_i$ is identical to the future target block, as shown in Fig. 3. The trade-off is an increase in memory usage due to storing backups for possibly more efficient copy update encoding in the future.
- **N - Next Block to Process:** Sometimes, it is more advantageous to perform block-wise updates in a certain order other than from 1 to $n$. Therefore, an action that moves to an arbitrary block is introduced. This action is always performed regardless of whether **M** or **MB** is chosen.

A CAV might have multiple ECUs to update in parallel, each with distinct Software in a real-world scenario. For clarity, the setting above focuses on a single ECU firmware, but the principle extends to multi-ECU systems by running separate agents or employing a multi-agent RL framework. The approaches described here remain valid at scale, highlighting the need for efficient decision-making to minimize bandwidth usage (especially over cellular networks) and ensure reliable updates without compromising ongoing vehicular functions.

## *C. Markov Decision Process (MDP) Formulation*

To systematically capture the OTA update sequence, we model it as a Markov Decision Process (MDP) $\mathcal{M} = (S, A, P, R, \gamma)$ :

*1) **State Space** $S$:* Each state $s \in S$ encodes the essential information needed for decision-making, such as:

- **Current Block Index:** An integer pointer indicating which block (for example, $o_i$) the agent focuses on.

- **Unprocessed Blocks:** A list or set of block indices that have not been processed and updated to their respective target blocks.

- **Action Taken:** In partially observable MDP environments, observation alone is not enough to recover the agent's current state. Inspired by [34], a list of past actions taken can be used to help in recovering the state.

- **Completion State:** Whether all blocks have been updated. If so, the current episode needs to be terminated.

The dimensionality of $S$ depends on the size of the Software, the number of backups, and additional variables reflecting the network's or the ECU's constraints. The state must be carefully engineered to maintain tractable RL learning.

*2) **Reward Function** $R$:* The reward model plays a key part in the evaluation of outputs generated by the policy network. Its primary function is to give a score to each output based on how close it is to the desired objective of the OTA update. In this case, it can take advantage of resource costs, such as cumulative runtime memory usage, transmission overhead, or partial patch size.

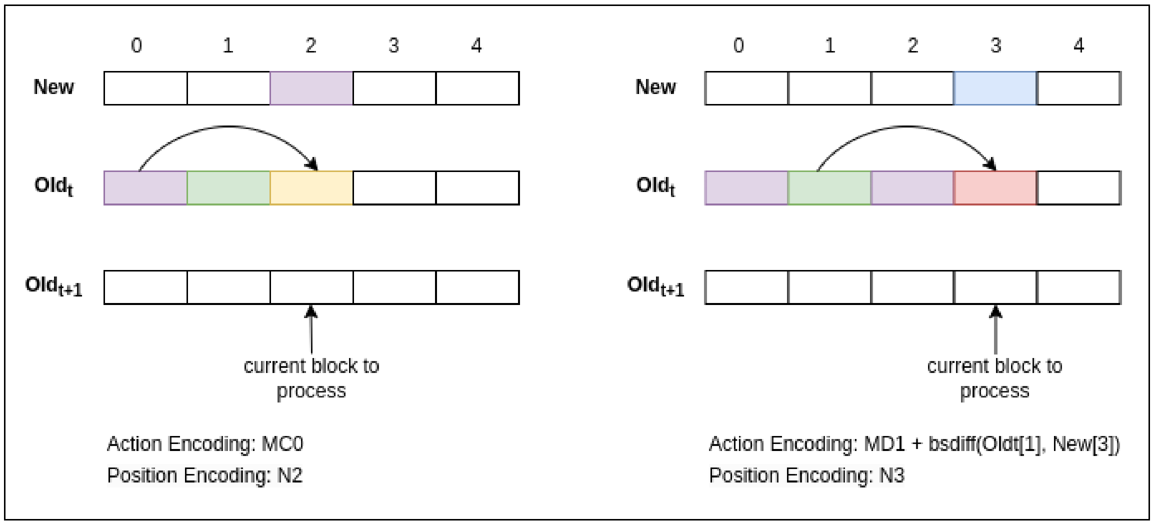


Fig. 2: Proposed Block-wise Update Action M. Left part is copy update (MC) and right part is delta update (MD), in which the "closest" block in the old block list is chosen to be delta-ed against the target new block.

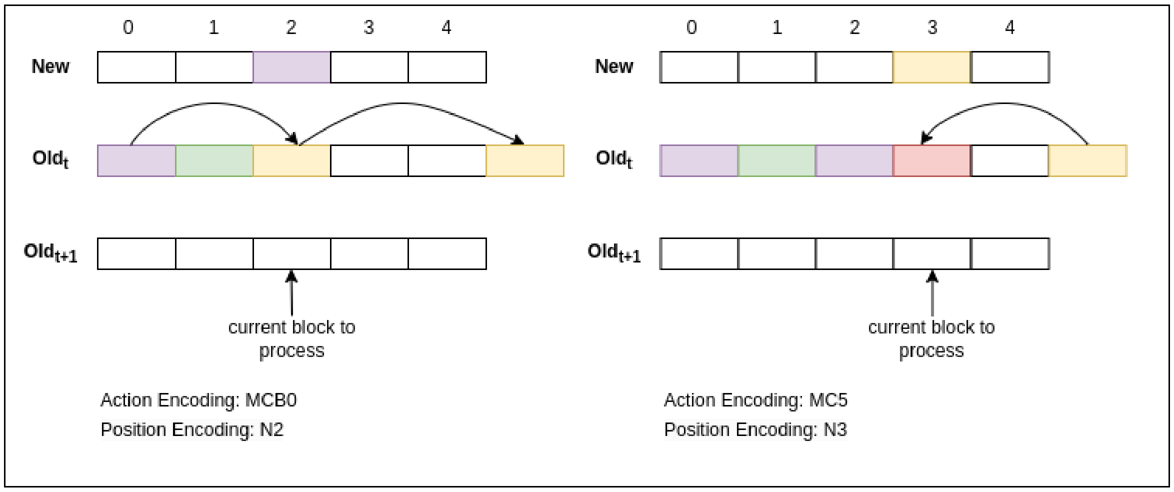


Fig. 3: Proposed Block-wise Update Action MB. Sometimes it is more beneficial to make a temporary copy of old block $o_i$ before overwriting it with updated block, which leads to a more efficient copy update in the future.

*3) **Action Space** $A$ :* The agent selects actions from $\{M, MB\}$ on chosen block $i \in \{1..n\}$. While simplistic in enumeration, the effect of each action depends heavily on the underlying Software blocks. For instance, "modify" might involve a small patch if two blocks are similar or a large patch otherwise.

*4) **Transition Dynamics** $P$ :* The environment transitions deterministically (or near-deterministically) from $s_t$ to $s_{t+1}$ once the agent executes an action $a_t$. Examples include:

- **Modify**: When the modification is performed, the current block is processed and removed from the list of yet-to-be-processed blocks. The current block chosen is overwritten with the updated block.

- **Backup**: The new state has an additional memory cost of exactly one block compared to the previous state. For ease of implementation, we simply move the block to the back of the current block list.

- **Choice of Block**: The processing block index changes to the one chosen, which has a constant cost ('N' + length of block index).

*5) **Policy Representation**:* Deep neural networks are commonly used for representing policies that feature high-dimensional state space. Proximal Policy Optimization (PPO), for instance, employs an actor-critic [35] network architecture usually with a shared backbone.

*6) **Objective**:* The agent is targeted at:

- Minimizing the final update patch size through a deliberate choice of update sequence.
- Achieve a balance between update patch size and runtime memory cost (maximum of a list of blocks residing in memory + backup blocks)

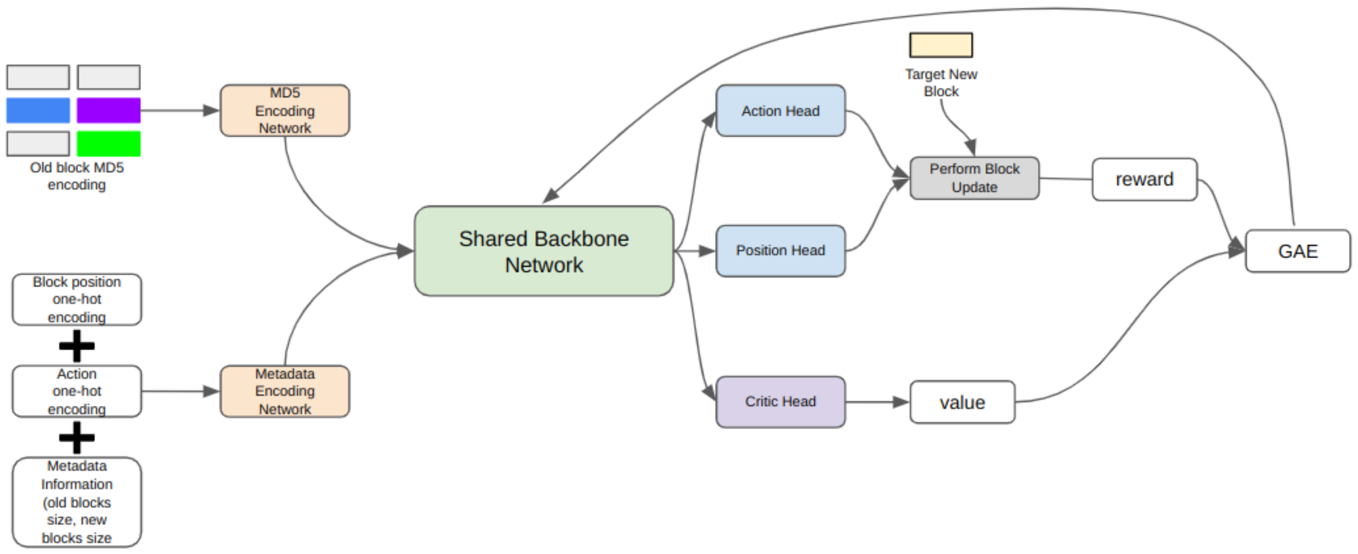


Fig. 4: Demonstration of the PPO network models used.

## IV. Methodology

### A. *Overview of the System Model*

The system model represents the OTA update as a Markov Decision Process (MDP) optimized using Proximal Policy Optimization (PPO). The process transforms old software blocks $O$ into new software blocks $N$, minimizing encoding size and memory cost.

1) ***Components***:

- **Software Blocks:** Partitioned into fix-sized blocks $O = \{o_1, o_2, \ldots, o_n\}$ and $N = \{n_1, n_2, \ldots, n_m\}$, enabling block-level updates. Here, we let each block be a small multiple of 4K (4096 bytes).
- **RL Environment:** Defines the state $s_t$, action space $A$, and reward function $R(s_t, a_t)$.
- **RL Agent:** Learns an optimal policy $\pi_\theta(a_t \mid s_t)$ for choosing update actions while minimizing update patch size and memory cost.

### B. *Step-by-Step Workflow*

1) ***Data Preparation***:

- **Data Partitioning:** Divide the old software blocks $O$ and new software blocks $N$ into fixed-size blocks:

$$O = \{o_1, o_2, \ldots, o_n\}, \quad N = \{n_1, n_2, \ldots, n_m\}$$

- **Block size**: $B$ is a hyperparameter (e.g., 64 KB ) and is a multiple of 4K, selected to balance granularity and efficiency.
- **MD5 Computation:** Since each block can be a multiple of 4K bytes, the input size would be infeasible for the neural network. As a result, we compute MD5 hashes for each block as a way to represent them in reduced dimension:

$$H(o_i) = \mathrm{MD5}(o_i), \quad H(n_j) = \mathrm{MD5}(n_j)$$

- Store additional metadata, including block indices, sizes, and a list of un-processed blocks

### C. *Environment Creation*

The custom Gym environment $E$ (OTAEnv) models the OTA process. It includes observations, step functions, rewards, and termination criteria.

1) ***Observation*** $(o_t)$ : $o_t = \{$ block mask, action one-hot encoding, size of old software blocks, size of new software blocks $\}$

- **Block Mask:** Indicates which blocks have already been processed.
- **One-hot Encoding of Action:** Tracks current processing action in one-hot vector ($1 \times 2$).
- **Size of old and new software blocks:** These are the metadata information to assist the agent in adapting to variable lengths of software blocks in old and new versions. For example, if the old software blocks have a larger size, then after updating the relevant blocks, the excessive blocks can be truncated.

2) ***Step Functions*** :

$$A = \{M, MB\} \cup \{1..n\}$$

- $M$ **(Modify)**: Update the current block either by direct copy or delta operation.
- $MB$ **(Modify + Backup):** Update the block and store a backup for rollback.
- $N$ **(next block to process):** Choose the next to block to process from 1 to $n$.

*3) **Reward Function** $(R)$ :* The reward $r_t$ at timestep $t$ balances encoding size and memory usage:

$$r_t = \frac{C_{\max} - (\alpha \cdot \text{ encoding cost } + \beta \cdot \text{ memory cost })}{C_{\max}}$$

Where: $C_{\max}$ : Normalization constant. $\alpha, \beta$ : Scaling factors for balancing encoding size and memory cost.

*4) **Termination Criteria**:* The episode terminates when all blocks are processed $(U_t = \emptyset)$ or a predefined maximum step limit is reached.

## *D. **RL Agent***

The RL agent is implemented using PPO (shown in Fig. 4), which optimizes the policy for selecting actions and blocks to process. In the above figure, we presented the overall neural network architecture of the reinforcement learning framework for block-wise OTA updates, where old block MD5 encodings, block metadata-like block sizes or positions, and action encoding are given as input. Among them, the former two types are fed separately into the encoding networks in the form of MD5 "Encoding Network" and "Metadata Encoding Network". These encodings are subsequently combined and fed into a shared backbone network that outputs via three specialized heads: the Action Head predicts the optimal action, the Position Head chooses the target block, and the Critic Head estimates the value function. The "Perform Block Update" module calculates rewards based on the selected action and target block for GAE to do the training of the reinforcement learning agent.

*1) **Actor-Critic Neural Network**:* The Actor outputs a probability distribution over actions:

$$\pi_\theta(a_t \mid s_t) = P(a_t \mid s_t; \theta)$$

The Critic estimates the value of the current state:

$$V(s_t; \phi) = \mathbb{E}_{\pi_\theta}\left[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}\right]$$

*2) **Advantage Estimation**:* Generalized Advantage Estimation (GAE) is used:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots$$

Where:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

*3) **Policy Optimization**:* PPO maximizes the expected cumulative reward $L(\theta)$ using the clipped objective:

$$L(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \operatorname{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

## *E. **Training and Evaluation***

*1) **Training**:* - The agent interacts with the environment over multiple episodes to learn an optimal policy between two versions of software.

- Periodically update the policy parameters $\theta$ and $\phi$.

*2) **Evaluation**:* The learned policy is tested on unseen versions of software blocks, and is evaluated based on update patch size (total encoding cost) and runtime memory overhead.

**Algorithm 1** summarizes the overall workflow of the proposed reinforcement learning framework for block-wise OTA updates. It describes how the implementation will be done step by step: partitioning software blocks, initializing the environment and PPO agent, training by episodic interactions, and evaluation of the learned policy on unseen software pairs. In such a way, the structured approach will ensure efficient optimization of OTA updates with respect to encoding size, memory overhead, and overall performance.

## *F. **Complexity Consideration***

The proposed OTA update framework needs to ensure scalability and computational efficiency. Some key considerations are as follows:

*1) **Scalability in Software Size**:* The framework uses block-level partitioning and maintains a table for tracking unprocessed blocks, ensuring linear complexity:

$$\mathcal{O}(\text{ processing complexity }) = \mathcal{O}(n+m)$$

## *G. **Runtime Complexity***

The RL agent is trained offline based on the assumption that the backend server has ample computational and memory resources. Policy inference during deployment is lightweight, mitigating real-time computational overhead:

$$\mathcal{O}(\text{ inference complexity }) = \mathcal{O}(f(s_t))$$

and can be done linearly with respect to the total number of software blocks. Here, $f(s_t)$ is the forward pass of the policy network (actor) that computes the action probabilities based on the current state $s_t$

TABLE II: Hyperparameters Used

| **Name** | **Value** |
|---|---|
| Learning Rate | 1e-4 |
| $\gamma$ | 0.99 |
| $\epsilon$ | 0.1 |
| $\alpha$ | 0.1 |
| $\beta$ | 0.01 |
| Block Size | 64 KB |
| Optimizer | ADAM |

## V. Experiment

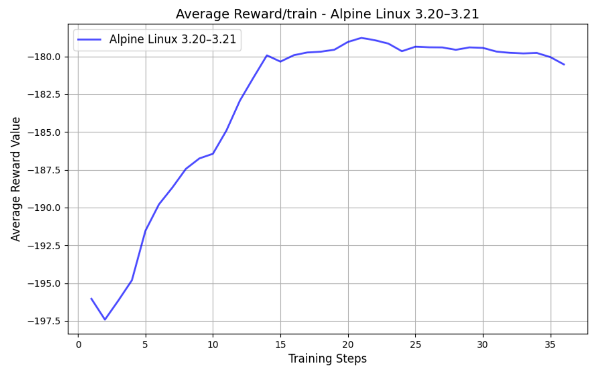
In our experiments, we chose Alpine Linux image versions 3.18.0, 3.19.1, 3.20.0, and 3.21.0, which mimic the full system update process for one ECU. We set the maximum number of episodes to 1000 and the network update timestep to 200. We used Pytorch 2.5.1 with CUDA 12.4 support, and the experiment was conducted on a workstation with a 14-core i7-12700H processor and RTX 3060 graphics card.

The RL model was trained using a proximal policy optimization (PPO) algorithm. A custom Actor-Critic architecture was employed for the PPO agent, with hyperparameters

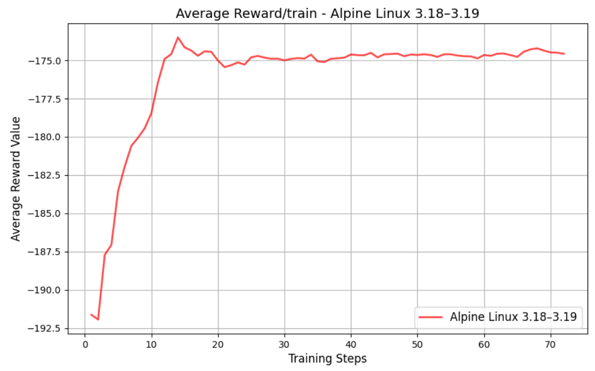
**Algorithm 1** Proximal Policy Optimization (PPO) for Block-wise OTA Updates

```
 1: Input:
 2:    Old software blocks $O = \{o_1, \dots, o_n\}$, New software blocks $N = \{n_1, \dots, n_m\}$
 3:    PPO hyperparams: Learning rate $\alpha$, discount $\gamma$, clip $\epsilon$
 4:    Max episodes $E_{\max}$, max timesteps $T_{\max}$
 5: Output: Optimized OTA update policy network $\pi_\theta(a_t \mid s_t)$
 6: function PartitionSoftware($O, N$)
 7:     Divide $O$ and $N$ into fixed-size blocks.
 8:     Compute MD5 hashes for all blocks: $H(o_i)$, $H(n_i)$
 9: end function
10: function InitializePPOAgent
11:     Set up actor network $\pi_\theta(a \mid s)$ and critic $V_\phi(s)$.
12: end function
13: function InitializeEnvironment($O, N$)
14:     Load firmware blocks and metadata into environment $E \leftarrow \text{OTAEnv}(O, N)$.
15:     Define initial state $s_0 = \{b_0, \text{mask}_0, c_0, m_0\}$.
16: end function
                                                        ▷ Training Loop
17: for episode $e = 1$ to $E_{\max}$ do
18:     Reset environment $E$, get $s_0$.
19:     for timestep $t = 1$ to $T_{\max}$ do
20:         Observe state $s_t$.
21:         Sample action $a_t, b_t \sim \pi_\theta(a \mid s_t)$.
22:         Execute $a_t$ on block $b_t$ in $E$:
                $s_{t+1}, r_t \leftarrow \text{StepEnv}(E, s_t, a_t, b_t)$
23:         Store $(s_t, a_t, r_t, s_{t+1})$ in buffer $B$.
24:         $s_t \leftarrow s_{t+1}$.
25:     end for
26:                                                 ▷ Perform PPO updates
27:     while buffer $B$ not empty do
28:         Compute advantages $\hat{A}_t$ via GAE: $\hat{A}_t = \delta_t + \gamma\lambda\,\delta_{t+1} + \dots$
29:         TD error: $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$.
30:         Optimize PPO objective:
                $L(\theta) = \mathbb{E}_t\big[\min(r_t(\theta)\,\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t)\big]$
31:     end while
32: end for
                                                     ▷ Evaluation Phase
33: function EvaluatePolicy($\pi_\theta(a \mid s)$)
34:     Freeze $\pi_\theta$.
35:     Test on unseen pairs $(O', N')$.
36:     Measure total encoding size, memory overhead, and reward.
37: end function
```

outlined in Table II. The training involved a dense reward function balancing encoding size, memory cost, and runtime constraints. The environment processed blocks sequentially for each image pair, guided by actions sampled from the policy network and on the desired block sampled from the block encoding head. The agent enhanced decision-making to reduce overall patch size and runtime memory consumption while guaranteeing precise updates.



(a) Average reward and metrics in training on generating updating patch between version 3.20 and 3.21 of Alpine Linux image.

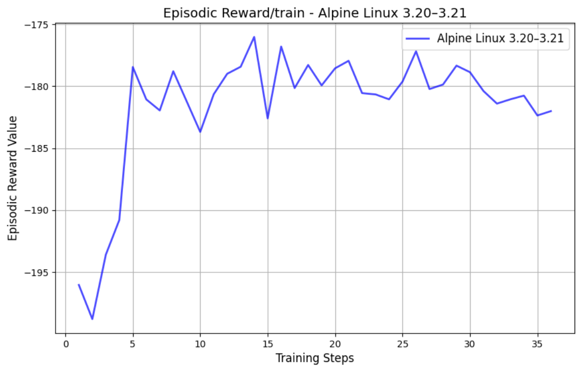


(b) Average reward and metrics in training on generating updating patch between version 3.18 and 3.19 of Alpine Linux image.

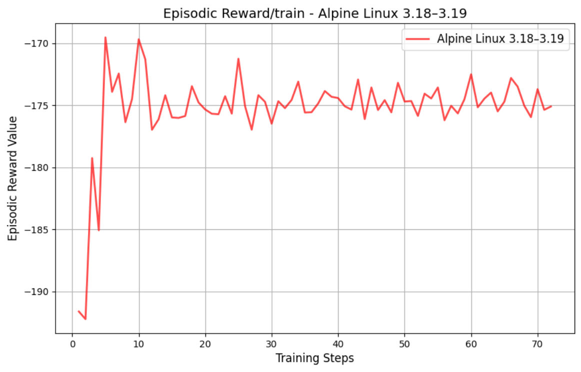
Fig. 5: Average reward and metrics in training on generating updating patch.

Fig. 5 illustrates the average reward and metrics in training and convergence behavior to generate the OTA update patch for the selected image pairs. Specifically, Fig. (5a) presents the results for Alpine Linux versions 3.20.0- 3.21.0 and Fig. (5b) shows the results for version 3.18.0 and 3.18.1. We examined the evolution of average rewards throughout several episodes in Fig. 5 to better understand the agent's learning stability. The average reward for the 3.20–3.21 version pair shown in Fig. (5a) demonstrates a consistent increase throughout the first 20 episodes, eventually settling around close to -180 with slight variation. This shows steady policy improvement and robust handling of moderately challenging OTA scenarios. In addition, the average reward for the 3.18-3.19 pair increases quickly during the first ten episodes, stabilizing at about -

175 as shown in Fig. (5b), which further supports the lower complexity of the update task. These outcomes highlight the PPO agent's capacity to attain high policy stability while effectively adjusting to varying job complexity levels.



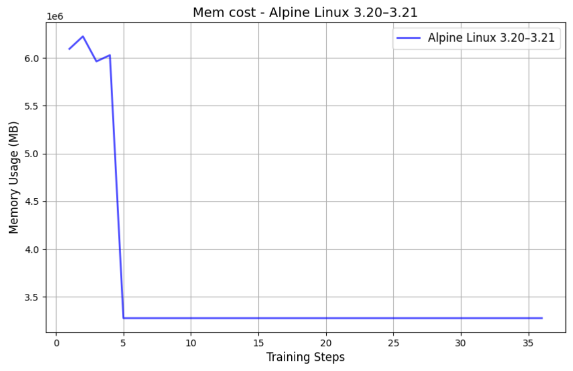
(a) Episodic reward comparison during training for Alpine Linux versions 3.20–3.21. This plot illustrates the episodic performance improvement over successive training steps.



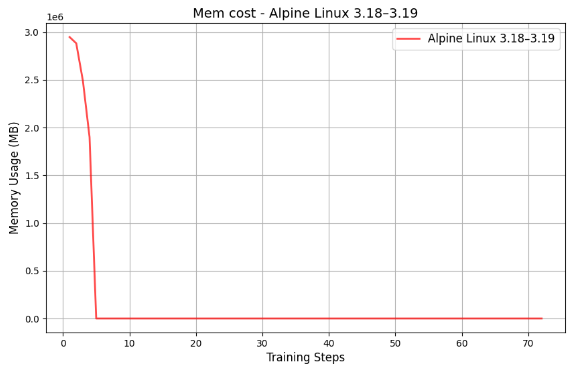
(b) Episodic reward trends for Alpine Linux versions 3.18–3.19 during training. The chart shows the model's ability to achieve episodic gains and stabilize over time.

Fig. 6: Episodic reward comparison during training.

We illustrate the episodic reward performance improvement over successive training steps in Fig. 6 and demonstrate the Alpine Linux version pair's episodic reward trends in (3.20–3.21 and 3.18–3.19) by showing how well the PPO agent can adjust to OTA update jobs of different complexity. The agent begins with a low reward of roughly -195 for version 3.20-3.21, as illustrated in Fig. (6a), but it rapidly improves over the following 10 to 15 episodes, stabilizing between -175 and -180. This behavior demonstrates the moderate difficulty of the optimization job, which requires the agent to manage the trade-off between exploration and exploitation in order to achieve optimal performance. In contrast, for version pair 3.18–3.19, shown in Fig. (6b), the agent reaches convergence much more quickly, stabilizing within 10 episodes at an average reward of approximately -175. The faster convergence and reduced reward fluctuations highlight the agent's efficiency in handling less complex update scenarios. To understand



(a) Memory cost analysis for Alpine Linux versions 3.20–3.21. The graph reflects resource utilization and memory efficiency during the training phases.



(b) Memory cost analysis for Alpine Linux versions 3.18–3.19. The graph reflects resource utilization and memory efficiency during the training phases.

Fig. 7: Average memory usage in training on generating update patch.

the average memory usage in training to generate the update patch, we performed a memory cost analysis for Alpine Linux versions (3.20-3.21 and 3.18–3.19) in Fig. 7, where the experiment reflects resource utilization and memory efficiency during the training phases. The experiment further highlights the proficiency of the PPO agent in optimizing run-time overhead. The memory utilization for the 3.20–3.21 pair, as depicted in Fig. (7a), is high at first, at around 6 MB (6291456 bytes), but it rapidly decreases to stable at about 3.5 MB (3538944 bytes) after only five training steps. This quick optimization demonstrates how well the agent can handle memory in relatively complex OTA update tasks. Similarly, for the 3.18–3.19 pair, as shown in Fig. (7b), the update patch size is optimized, as evidenced by the memory usage beginning at a lower 3 MB (3145728 bytes) and stabilizing below 1 MB 1048576 bytes) after five steps, reflecting the more straightforward nature of the update process and optimization of the update patch size. In both cases, the agent successfully minimizes memory usage early in training, demonstrating adaptability and efficiency.

Together, the findings show how flexible our proposed PPO agent is while managing OTA update tasks of various complexity. The agent consistently converges in both average and episodic rewards, and more straightforward version pairings show faster stabilization. Furthermore, the memory optimization outcomes support the agent's ability to minimize runtime overhead during the first training phase. These results confirm that the suggested reinforcement learning method is a scalable and effective way to maximize OTA updates in various situations.

## VI. Conclusion and Future Work

In this paper, we proposed a new way of framing the full system OTA update problem as MDP process and demonstrated the potential of deep reinforcement learning in optimizing the update objectives under constrained resources, such as memory overhead and size of over-the-air update patch. We developed a proof-of-concept environment, OTAEnv, and a sample training script to validate its effectiveness. In the future, we plan to evaluate the scalability of our OTA approach using BlueICE [36], a customizable simulation platform with distributed co-simulation capabilities, enabling analysis of its performance in complex and diverse autonomous driving scenarios where software OTA update is a critical factor.

## References

[1] M. M. Rana and K. Hossain, "Connected and autonomous vehicles and infrastructures: A literature review," *International Journal of Pavement Research and Technology*, vol. 16, no. 2, pp. 264–284, 2023.

[2] S. Halder, A. Ghosal, and M. Conti, "Secure over-the-air software updates in connected vehicles: A survey," *Computer Networks*, vol. 178, p. 107343, 2020.

[3] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, Chen, and Q. Alfred, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 385–396.

[4] A. Bhattacharjee, H. Mahmood, S. Lu, N. Ammar, A. Ganlath, and W. Shi, "Edge-assisted over-the-air software updates," in *2023 IEEE 9th International Conference on Collaboration and Internet Computing (CIC)*, 2023, pp. 18–27.

[5] Z. Bowden and J. Chandler, "Current implementations of over-the-air updates," National Surface Transportation Safety Center for Excellence, Tech. Rep., 2024.

[6] V. Nikic, D. Bortnik, M. Lukic, D. Danilovic, and I. Mezei, "Comparisons of firmware delta updates over the air using wlan and lpwan technologies," in *2022 30th Telecommunications Forum (TELFOR)*. IEEE, 2022, pp. 1–4.

[7] S. El Jaouhari and E. Bouvet, "Secure firmware over-the-air updates for iot: Survey, challenges, and discussions," *Internet of Things*, vol. 18, p. 100508, 2022.

[8] N. H. Hussein, C. T. Yaw, S. P. Koh, S. K. Tiong, and K. H. Chong, "A comprehensive survey on vehicular networking: Communications, applications, challenges, and upcoming research directions," *IEEE Access*, vol. 10, pp. 86 127–86 180, 2022.

[9] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware over-the-air programming techniques for iot networks-a survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–36, 2021.

[10] X. Zhang, Y. Wang, Y. Su, H. Zuo, J. Huang, and L. Kong, "An ota upgrade differential compression algorithm based on suffix array induced sorting and bsdiff methods," *Applied Sciences*, vol. 14, no. 2, p. 544, 2024.

[11] J. Kwon, Y. Efroni, C. Caramanis, and S. Mannor, "Reinforcement learning in reward-mixing mdps," *Advances in Neural Information Processing Systems*, vol. 34, pp. 2253–2264, 2021.

[12] Z. Li, G. Qin, Y. Liang, Danzengouzhu, and L. Yang, "Bsdiff difference algorithm based on lzma2 for in-vehicle ecus," in *International Conference on Computing, Control and Industrial Engineering*. Springer, 2023, pp. 719–725.

[13] H. Tan, W. Xia, X. Zou, C. Deng, Q. Liao, and Z. Gu, "The design of fast delta encoding for delta compression based storage systems," *ACM Transactions on Storage*, 2024.

[14] S. Wu, Z. Tu, Y. Zhou, Z. Wang, Z. Shen, W. Chen, W. Wang, W. Wang, and B. Mao, "Fastsync: a fast delta sync scheme for encrypted cloud storage in high-bandwidth network environments," *ACM Transactions on Storage*, vol. 19, no. 4, pp. 1–22, 2023.

[15] J. Gong and T. Chen, "Does configuration encoding matter in learning software performance? an empirical study on encoding schemes," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 482–494.

[16] M. Alkwiefi *et al.*, "M2m protocols: An overview on lwm2m and xmpp machine-to-machine protocols in iot context," in *2024 IEEE/ACIS 24th International Conference on Computer and Information Science (ICIS)*. IEEE, 2024, pp. 209–215.

[17] A. Almheiri and Z. Maamar, "Iot protocols–mqtt versus coap," in *Proceedings of the 4th International Conference on Networking, Information Systems & Security*, 2021, pp. 1–5.

[18] V. M. et al., "Human-level control through deep reinforcement learning," *Nature*, 2015.

[19] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning." Association for Computing Machinery, 2016, p. 50–56.

[20] Y. Gu, Y. Cheng, C. P. Chen, and X. Wang, "Proximal policy optimization with policy feedback," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 7, pp. 4600–4610, 2021.

[21] E. H. Bouzidi, A. Outtagarts, and R. Langar, "Deep reinforcement learning application for network latency management in software defined networks," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[22] S. Padakandla, "A survey of reinforcement learning algorithms for dynamically varying environments," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–25, 2021.

[23] L. Luo, L. Sheng, H. Yu, and G. Sun, "Intersection-based v2x routing via reinforcement learning in vehicular ad hoc networks," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 5446–5459, 2021.

[24] A. Ahadi-Sarkani and S. Elmalaki, "Adas-rl: Adaptive vector scaling reinforcement learning for human-in-the-loop lane departure warning," in *Proceedings of the First International Workshop on Cyber-Physical-Human System Design and Implementation*, 2021, pp. 13–18.

[25] W. Li, H. Cui, T. Nemeth, J. Jansen, C. Uenluebayir, Z. Wei, L. Zhang, Z. Wang, J. Ruan, H. Dai *et al.*, "Deep reinforcement learning-based energy management of hybrid battery systems in electric vehicles," *Journal of Energy Storage*, vol. 36, p. 102355, 2021.

[26] K. Fizza, N. Auluck, A. Azim, M. A. Maruf, and A. Singh, "Faster ota updates in smart vehicles using fog computing," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2019, pp. 59–64.

[27] C. E. Andrade, S. D. Byers, V. Gopalakrishnan, E. Halepovic, D. J. Poole, L. K. Tran, and C. T. Volinsky, "Scheduling software updates for connected cars with limited availability," *Applied Soft Computing*, vol. 82, p. 105575, 2019.

[28] H. U. Park, J. Jeong, and P. Mah, "Non-invasive rapid and efficient firmware update for wireless sensor networks," in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, 2014, pp. 147–150.

[29] G. Kornaros, O. Tomoutzoglou, D. Mbakoyiannis, N. Karadimitriou, M. Coppola, E. Montanari, I. Deligiannis, and G. Gherardi, "Towards holistic secure networking in connected vehicles through securing canbus communication and firmware-over-the-air updating," *Journal of Systems Architecture*, vol. 109, p. 101761, 2020.

[30] S. Yeasmin, A. Haque, and A. Sayegh, "A novel and failsafe blockchain framework for secure ota updates in connected autonomous vehicles," *Vehicular Communications*, vol. 43, p. 100658, 2023.

[31] A. Anwar, N. Chakraborty, and M. Zulkernine, "Secure ota software updates for connected vehicles using lorawan and blockchain," in *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 2024, pp. 1067–1076.

[32] A. W. Malik, A. U. Rahman, A. Ahmad, and M. M. D. Santos, "Over-the-air software-defined vehicle updates using federated fog environment," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 5078–5089, 2022.

[33] S. Bagchi, V. Aggarwal, S. Chaterji, F. Douglis, A. El Gamal, J. Han, B. J. Henz, H. Hoffmann, S. Jana, M. Kulkarni *et al.*, "Vision paper: Grand challenges in resilience: Autonomous system resilience through design and runtime measures," *IEEE Open Journal of the Computer Society*, vol. 1, pp. 155–172, 2020.

[34] P. Zhu, X. Li, and P. Poupart, "On improving deep reinforcement learning for pomdps," *ArXiv*, vol. abs/1704.07978, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:2310429

[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[36] Y. He, H. Chen, and W. Shi, "An advanced framework for ultra-realistic simulation and digital twinning for autonomous vehicles," 2024. [Online]. Available: https://arxiv.org/abs/2405.01328