The American Journal of Applied Sciences
ISSN 2689-0992 | Open Access

TYPE Original Research
PAGE NO. 23-33

Check for updates

AI-Enabled Software-Defined Vehicles: An Intelligent Cloud-Edge Architecture for Adaptive, Secure, and Resilient Automotive Software Systems

Srikanth Puram

Independent Researcher, Mobile and Embedded Software Architecture
Novi, Michigan, USA

OPEN ACCESS

SUBMITTED 05 January 2024
ACCEPTED 14 February 2024
PUBLISHED 30 March 2024
VOLUME Vol.06 Issue 03 2024

CITATION
Puram, S. (2024). AI-Enabled Software-Defined Vehicles: An Intelligent Cloud-Edge Architecture for Adaptive, Secure, and Resilient Automotive Software Systems. The American Journal of Applied Sciences, 6(03), 23–33. Retrieved from https://theamericanjournals.com/index.php/tajas/article/view/8217

Abstract: Software-defined vehicles (SDVs) are shifting automotive engineering from fixed electronic-control-unit deployments toward continuously evolving cloud-connected, edge-compute, and AI-assisted software platforms. This transition enables adaptive features, predictive diagnostics, cybersecurity monitoring, over-the-air software updates, and data-driven user experiences; however, it also increases architectural complexity, update risk, attack surface, and operational uncertainty. This article proposes an AI-enabled cloud-edge architecture for adaptive, secure, and resilient automotive software systems. The architecture integrates centralized and zonal compute domains, Android-based embedded application services, secure cloud-to-edge update orchestration, edge AI inference for anomaly and risk scoring, zero-trust access enforcement, software-bill-of-materials-aware package validation, and recovery mechanisms for interrupted update and application workflows. Unlike conventional SDV reference models that separately discuss AI, OTA updates, cybersecurity, and resilience, the proposed architecture treats them as a unified lifecycle problem: software must be selected, validated, deployed, monitored, recovered, and improved across vehicle, edge, and cloud layers. The paper defines the system layers, data/control flows, AI model placement strategy, threat model, update workflow, resilience state

machine, and evaluation metrics. The proposed framework is aligned with automotive software-update engineering, cybersecurity engineering, secure software development, AI risk-management, and cloud-edge security practices available through March 2024. The result is a practical technical architecture intended for next-generation automotive and embedded mobility platforms requiring reliable software evolution under safety, security, latency, and resource constraints.

Keywords: software-defined vehicles, Android Automotive, edge AI, TensorFlow Lite, cloud-edge orchestration, OTA updates, cybersecurity, zero trust, resilience engineering, predictive diagnostics, automotive software architecture.

I. Introduction

The automotive industry is undergoing a fundamental architectural transition. Vehicles that were historically organized around hardware-specific electronic control units are increasingly evolving into software-defined platforms where centralized compute, zonal controllers, cloud connectivity, continuous software delivery, and artificial intelligence determine much of the vehicle experience. This shift allows manufacturers to deliver post-sale features, safety improvements, energy optimization, diagnostics, personalization, and software corrections without replacing hardware. It also creates a new engineering challenge: the vehicle is no longer a static embedded system, but a distributed software ecosystem operating across vehicle, edge, and cloud environments.

The complexity of this ecosystem is not limited to autonomous-driving functions. In production SDVs, software delivery involves package validation, application lifecycle control, network-aware deployment, data synchronization, feature flags, cybersecurity monitoring, diagnostic telemetry, rollback policy, user consent, and recovery after power-state changes. A software update can involve cloud services, edge gateways, Android-based application layers, middleware, vehicle services, and multiple compute domains. If any stage fails, the system must preserve operational continuity and avoid ambiguous software state.

AI can improve this lifecycle when it is applied as an architectural capability rather than as an isolated model.

Lightweight on-device inference can classify update risk, detect anomalous telemetry, prioritize diagnostic events, predict component degradation, and assist recovery decisions. Cloud AI can train models using fleet-scale data and perform heavier analytics. The central design question is how to balance AI placement across vehicle, edge, and cloud layers while meeting latency, privacy, memory, compute, safety, and cybersecurity requirements.

This paper proposes an AI-enabled cloud-edge architecture for SDVs that unifies adaptive intelligence, secure update workflows, application/data security, and resilience mechanisms. The goal is not to replace automotive safety engineering or regulatory compliance; rather, it is to define a practical software architecture that can support continuously evolving vehicle software while maintaining integrity, observability, and recoverability.

The contributions of this paper are fourfold. First, it presents a layered SDV architecture connecting cloud services, edge intelligence, Android-based embedded application services, zonal compute, and vehicle signals. Second, it defines an AI placement strategy for vehicle, edge, and cloud inference. Third, it introduces a risk-adaptive update and application-data workflow using AI-assisted validation and zero-trust controls. Fourth, it proposes a resilience state model and evaluation metrics for update correctness, security, latency, and recovery behavior.

II. Technical Background

SDV architectures differ from traditional automotive software architectures because functionality is increasingly decoupled from fixed hardware nodes. Central compute and zonal architectures consolidate software services and reduce duplication, but they require stronger orchestration, isolation, dependency management, and validation. Prior research has described the transition toward SDVs, dynamic service-oriented architectures, resource allocation, virtualization, and software risk optimization [1]-[6].

Software update engineering has also become a formal automotive concern. ISO 24089:2023 specifies requirements and recommendations for software

update engineering for road vehicles, while UN Regulation No. 156 addresses software update and software update management systems [7], [10]. These practices are closely connected with cybersecurity engineering under ISO/SAE 21434:2021 and UN Regulation No. 155, which define risk-management expectations for vehicle cybersecurity [8], [9].

AI introduces additional lifecycle and assurance requirements. The NIST AI Risk Management Framework identifies governance, mapping, measurement, and management activities for AI risk [12]. NIST Cybersecurity Framework 2.0 expands cybersecurity risk governance and can be applied to connected software ecosystems [13]. For embedded AI deployment, TensorFlow Lite Micro and MLPerf Tiny demonstrate the importance of latency, energy, accuracy, and resource-aware benchmarking in constrained environments [14], [15].

Android-based embedded environments add their own constraints. Android provides application lifecycle management, package installation mechanisms, sandboxing, and background execution models. Android Automotive extends Android into vehicle environments where power-state behavior, garage mode, shutdown preparation, and service recovery influence long-running workflows. Therefore, SDV software architecture must combine mobile application reliability, embedded systems recovery, cloud-native orchestration, and automotive cybersecurity principles.

III. Problem Definition

The central problem addressed by this paper is how to design an SDV software architecture that can continuously update, monitor, and adapt vehicle software while preserving security, resilience, and operational correctness. A naive cloud-connected architecture may deliver artifacts and collect telemetry, but it may not ensure that the right package reaches the right device at the right time under safe operating conditions. Similarly, an AI layer may detect anomalies, but it may not be connected to update policy, rollback decisions, or secure telemetry handling.

The problem has five dimensions:

- Adaptive intelligence: vehicle functions and services must react to telemetry, usage patterns, environmental conditions, and diagnostics without requiring constant cloud interaction.
- Secure software delivery: packages, manifests, configuration data, and AI models must be authenticated, integrity-checked, version-controlled, and protected against replay or downgrade attacks.
- Application and data security: in-vehicle applications, cloud APIs, local data stores, telemetry channels, and diagnostic artifacts must be governed by least privilege and zero-trust controls.
- Resilience and recovery: interrupted downloads, failed installations, power transitions, process restarts, and partial data synchronization must be recoverable without corrupting operational state.
- AI placement: inference should be placed on vehicle, edge, or cloud infrastructure according to latency, privacy, connectivity, memory, energy, accuracy, and safety criticality constraints.

IV. Proposed Cloud-Edge SDV Architecture

The proposed architecture consists of six layers: cloud intelligence layer, secure update and policy layer, edge AI validation layer, embedded Android application-service layer, zonal/vehicle-service layer, and observability/governance layer. Figural representation is intentionally omitted so that the framework can be implemented across multiple OEM and supplier stacks; the logical components are summarized in Table I.

Layer	Primary responsibility	Security and resilience controls
Cloud intelligence	Fleet analytics, model training, policy generation, campaign targeting, telemetry aggregation.	Model governance, secure data pipelines, access control, audit trails, AI risk management.
Secure update and policy	Manifest generation, package selection, version rules, rollback policy, feature eligibility.	Signed manifests, SBOM linkage, anti-replay rules, zero-trust service authorization.
Edge AI validation	Local risk scoring, anomaly detection, connectivity-aware routing, update deferral decisions.	Lightweight inference, threshold policies, input validation, confidence logging.
Android embedded application services	Application package handling, local data synchronization, consent UI, background scheduling, lifecycle recovery.	Package signature checks, sandboxing, encrypted local state, checkpointed recovery.
Zonal/vehicle services	Vehicle signal access, compute-domain coordination, diagnostic state, service abstraction.	Least-privilege APIs, gateway filtering, power-state constraints, safe-state fallback.
Observability and governance	Event logging, update outcome reporting, security monitoring, compliance evidence.	Tamper-evident logs, telemetry minimization, anomaly correlation, incident-response hooks.

A. Cloud Intelligence Layer

The cloud intelligence layer aggregates fleet-level telemetry, update outcomes, diagnostic trends, and software configuration metadata. It trains heavier models that cannot run efficiently in the vehicle, including fleet-level anomaly detection, predictive maintenance models, update campaign risk models, and security-event correlation models. The output of this layer is not only an AI prediction but also a policy artifact: which devices are eligible, which conditions must be satisfied, which package set should be installed, and what rollback action is allowed.

B. Secure Update and Policy Layer

The secure update and policy layer produces signed update manifests and deployment policies. A manifest should include package identity, target version, dependency graph, cryptographic hashes, signature expectations, rollback policy, required device profile, AI model version, and data-migration rules. A deployment policy should specify eligibility, timing windows, connectivity requirements, power conditions, and escalation thresholds. This layer prevents the vehicle from making update decisions based only on file names or server responses.

C. Edge AI Validation Layer

The edge AI validation layer runs lightweight models near the vehicle or on the embedded device. Its purpose is not to replace cryptographic validation; rather, it complements deterministic checks with risk scoring. For example, a model may classify an update attempt as unusual because the package version jump is atypical, retry count is high, telemetry reports prior failures for a similar device profile, or the device is in a degraded power/storage state. The output is a decision

recommendation such as proceed, defer, retry later, quarantine, or require cloud revalidation.

D. Android Embedded Application-Service Layer

The Android embedded application-service layer handles user-visible and service-level workflows. It manages package installation, local data persistence, background scheduling, application consent or notification flows, and recovery after process death or power-state transition. This layer should persist checkpoints for long-running workflows and verify each checkpoint after resume. It should also separate UI state from update-state logic so that application recovery does not depend on the user reopening a screen.

E. Zonal and Vehicle-Service Layer

The zonal and vehicle-service layer abstracts access to signals, sensors, actuator-adjacent services, diagnostics, power state, and compute-domain boundaries. AI or update workflows should not access vehicle signals directly without authorization. Instead, a gateway or service interface should enforce policy, normalize data, and record audit context. This design aligns with zero-trust principles and reduces unintended coupling between applications and vehicle resources.

V. AI Model Placement and Data Flow

AI placement should be determined by operational constraints. A model that predicts imminent mechanical degradation may require low latency and local signal access. A model that learns population-level trends across millions of events may require cloud scale. A model that scores update risk can operate in hybrid form: cloud training and calibration, edge or device inference at deployment time, and cloud feedback after rollout.

AI workload	Preferred placement	Reason	Key metrics
Update risk scoring	Edge/device inference with cloud-trained model	Requires local power, storage, version, and retry context during deployment.	p95 inference latency, false accept rate, false reject rate, deferral accuracy.
Predictive diagnostics	Hybrid edge/cloud	Local inference supports fast alerts; cloud retraining improves accuracy across fleet trends.	Recall, precision, mean time to detect, model drift.
Cyber anomaly detection	Hybrid	Local rules catch immediate anomalies; cloud correlation detects campaign-level behavior.	Detection latency, alert precision, escalation rate.
Driver/user personalization	Vehicle/device	Privacy and latency favor local inference with minimal cloud exposure.	Latency, privacy exposure, local storage size.
Large-scale model improvement	Cloud	Training and evaluation need aggregated data and scalable compute.	Training cost, drift reduction, validation coverage.

For embedded SDV use cases, TensorFlow Lite, TensorFlow Lite Micro, and similar lightweight runtimes are relevant because they enable inference under memory and energy constraints [14]. MLPerf Tiny-style benchmarking is useful because it evaluates trade-offs across accuracy, latency, and energy rather than

measuring model quality in isolation [15]. For vehicle software workflows, the relevant AI question is not simply whether a model is accurate; it is whether the model improves a system decision without violating latency, safety, privacy, or recovery constraints.

VI. Risk-Adaptive Secure Update Workflow

The proposed secure update workflow treats OTA delivery as a governed lifecycle rather than a download operation. A package is not eligible for installation until deterministic validation, policy validation, and AI-assisted risk validation all pass. This section defines the workflow and decision algorithm.

A. Workflow Stages

- Manifest acquisition: obtain signed update metadata from the trusted update service.
- Device profile collection: collect installed version, hardware profile, storage, power state, network state, and software-policy context.
- Deterministic validation: verify package identity, version, hash, signature, dependencies, and compatibility.
- AI-assisted risk validation: evaluate anomaly and risk signals using a lightweight local model.
- Policy decision: proceed, defer, quarantine, retry, or require cloud reauthorization.
- Staged installation: install only after preconditions are met, preserving the previous known-good version.
- Post-install verification: confirm expected package state, service availability, and health-check outcome.
- Telemetry feedback: report minimized outcome data for cloud learning and audit evidence.

B. AI-Assisted Decision Algorithm

Algorithm 1: Risk-adaptive update decision

Input: manifest M, package set P, device profile D, policy K, local model f, thresholds T Output: action in {INSTALL, DEFER, RETRY, QUARANTINE, CLOUD_REVALIDATE}

1: if not verify_signature(M) or not verify_hashes(P, M) then

2: return QUARANTINE

3: end if

4: if not compatible(P, D) or not authorized(M, K) then
5: return CLOUD_REVALIDATE

6: end if

7: x <- features(D.power, D.storage, D.network, D.version_drift, D.retry_count, D.rollback_history, P.size, M.criticality)

8: r <- f(x)

9: if r >= T.high then return CLOUD_REVALIDATE

10: if D.power_unsafe or D.storage_low then return DEFER

11: if transient_network_failure and retry_budget_available then return RETRY

12: return INSTALL

The algorithm intentionally combines deterministic gates with AI-assisted scoring. Cryptographic and compatibility failures are never overridden by an AI model. The model only influences risk-sensitive choices among otherwise valid actions. This makes the architecture more auditable and safer than black-box automated installation decisions.

VII. Security Architecture and Threat Model

The SDV security architecture must protect code, data, models, policies, and telemetry. Threats include package tampering, manifest replay, downgrade attacks, unauthorized configuration updates, compromised edge components, excessive telemetry collection, model drift, poisoned training data, and failure to rollback after installation errors. Table III maps representative threats to controls.

Threat	Impact	Control
Tampered package or manifest	Unauthorized code execution or configuration corruption.	Signed manifests, hash verification, package signatures, trusted update channel.
Replay or downgrade attack	Older vulnerable software restored without authorization.	Monotonic version policy, manifest timestamp, anti-replay token, rollback authorization.
Compromised edge node	Manipulated risk score or policy response.	Zero-trust service authentication, attested clients where available, cloud revalidation for high-risk actions.
Poisoned telemetry or model drift	Incorrect AI risk scoring or diagnostic prediction.	Data-quality checks, model versioning, drift monitoring, human-review thresholds.
Unauthorized vehicle-signal access	Privacy exposure or unsafe dependency on vehicle data.	Gateway policy, least-privilege APIs, audit logging, data minimization.
Interrupted installation	Ambiguous software state or failed service recovery.	Checkpointing, staged activation, post-install verification, rollback policy.

Zero trust is applied as a software architecture principle. The vehicle should not trust a cloud instruction solely because it originated from an internal network. The cloud should not trust telemetry solely because it came from a registered vehicle. Each request should carry identity, authorization, device state, and policy context. This is especially important when applications, data services, update clients, and AI models operate across organizational and supplier boundaries.

VIII. Resilience and Recovery Model

Resilience is the ability to preserve a known-good operational state despite interruptions, invalid inputs, or partial failures. For SDV software workflows, the recovery model should persist each validated checkpoint and avoid trusting in-memory state after suspend, reboot, or process death. The proposed recovery state machine includes Idle, Manifest Received, Downloading, Downloaded, Validated, Staging, Committed, Verifying, Completed, Failed Recoverable, Failed Non-Recoverable, and Cleanup Required.

The state machine supports four principles. First, progress is recorded only after validation. Second, every recovery action must be idempotent. Third, after resume or reboot, constraints must be re-evaluated rather than reused from stale memory. Fourth, rollback or cleanup must preserve the previous known-good version whenever the target state cannot be verified.

Failure scenario	Required recovery behavior	Observable evidence
Network loss during download	Resume only after partial file validation or restart download.	Artifact ID, byte range, hash status, retry count.
Suspend during staging	Reload checkpoint, validate session state, recreate if inconsistent.	Session ID, checkpoint state, power event timestamp.
Process death after commit	Verify installed version and required splits	Expected version, observed version, package
	before reporting success.	verification result.
Post-install health failure	Rollback or mark non-recoverable according to policy.	Health-check result, rollback action, failure class.
AI score exceeds risk threshold	Defer or request cloud revalidation rather than installing.	Risk score, features used, decision threshold.

IX. Evaluation Methodology

The architecture should be evaluated using controlled experiments rather than only descriptive comparison. Because vehicle-level deployment data may be unavailable in early design stages, the first evaluation phase can use a reference Android embedded environment, representative update packages, synthetic telemetry, and fault-injection scenarios. The goal is to measure correctness, latency, resource cost, security decision quality, and recovery behavior.

Metric	Definition	Target interpretation
Update validation latency	Time to complete manifest, hash, signature, compatibility, and AI risk checks.	Lower latency improves deployment windows; p95 is more important than average.
Risk false accept rate	Invalid or unsafe update classified as acceptable.	Should be minimized; cryptographic failures must be zero-tolerance.
Risk false reject rate	Valid update deferred or escalated unnecessarily.	Should be low enough to avoid excessive rollout delay.
Recovery correctness	Percentage of interrupted workflows restored to correct state.	Should approach 100 percent for tested interruption states.
Rollback detection time	Time between failed post-install verification and rollback/cleanup	Lower values reduce ambiguous state

	action.	duration.
Memory overhead	Peak additional memory used by validation and AI inference.	Must remain within embedded Android service budget.
Telemetry minimization ratio	Reduction between raw collected data and reported security/update evidence.	Higher values reduce privacy and bandwidth exposure.

A publishable evaluation can include four test groups. The first group validates normal update completion across device profiles. The second injects package and manifest faults. The third interrupts the workflow through network loss, process termination, storage pressure, and suspend/resume. The fourth evaluates AI risk scoring using labeled scenarios such as normal rollout, repeated retry, unusual version drift, storage-constrained device, and prior rollback history. The measurement protocol should report p50 and p95 latency, memory overhead, decision accuracy, and recovery outcome.

X. Comparative Analysis

Compared with a conventional update client, the proposed architecture adds policy-aware manifest processing, AI-assisted risk scoring, checkpointed recovery, and zero-trust service interaction. Compared with a cloud-only AI architecture, it reduces dependency on continuous connectivity and supports local decision-making during deployment windows. Compared with an edge-only architecture, it preserves fleet-level learning, model governance, and campaign optimization in the cloud. The design therefore balances three competing objectives: local autonomy, centralized governance, and secure lifecycle control.

Approach	Strength	Limitation	Proposed improvement
Conventional OTA client	Simple package delivery and installation.	Limited risk awareness and weak recovery modeling.	Add signed policy, checkpoint validation, and recovery state machine.
Cloud-only intelligence	Strong fleet analytics and centralized control.	Latency, connectivity, and privacy limitations.	Move risk scoring and urgent diagnostics to edge/device inference.
Edge-only intelligence	Low latency and local autonomy.	Limited fleet learning and governance.	Use cloud training, model versioning, and secure feedback loop.
Rule-only security	Auditable deterministic decisions.	Hard to capture evolving anomaly patterns.	Combine deterministic gates with bounded AI-assisted risk scoring.

XI. Discussion

A key design choice is keeping AI decision-making bounded. In the proposed architecture, AI does not approve unsigned code, bypass compatibility checks, or override cybersecurity policy. Instead, AI contributes risk estimation, anomaly detection, prioritization, and routing decisions. This distinction matters because automotive software systems require explainable control points and deterministic fail-safe behavior.

Another practical consideration is model lifecycle. AI

models used for update risk scoring or diagnostics are themselves software artifacts. They require versioning, validation, rollback, monitoring, and security controls. A model update should therefore pass through a similar software-governance pipeline as application packages. The model card or metadata should specify training scope, intended use, input features, threshold values, known limitations, and rollback strategy.

The architecture is also intentionally compatible with heterogeneous implementation choices. It can be applied to Android Automotive, Android-based embedded systems, Linux-based gateways, or mixed vehicle compute stacks. The essential requirements are signed policy artifacts, secure update validation, local inference capability, checkpointed workflow recovery, and auditable telemetry.

XII. Conclusion

This paper proposed an AI-enabled cloud-edge architecture for adaptive, secure, and resilient software-defined vehicles. The architecture integrates secure update orchestration, edge AI validation, Android-based embedded application services, zonal vehicle services, zero-trust controls, and checkpointed recovery. The framework treats software delivery, cybersecurity, AI inference, application lifecycle, and resilience as one integrated lifecycle rather than separate engineering activities.

The proposed model is suitable for SDV and embedded mobility environments where software must evolve after deployment while preserving security and operational continuity. Future work should implement the architecture in a reference Android Automotive or embedded Android environment, run controlled fault-injection tests, evaluate AI risk scoring using labeled update scenarios, and compare cloud-only, edge-only, and hybrid inference placement strategies under realistic latency and memory constraints.

References

1. D. Slama, A. Nonnenmacher, and T. Irawan, The Software-Defined Vehicle: A Digital-First Approach to Creating Next-Generation Experiences. Sebastopol, CA, USA: O'Reilly Media, 2023.
2. Z. Liu, W. Zhang, and F. Zhao, "Impact, challenges and prospect of software-defined vehicles," Automotive Innovation, vol. 5, no. 2, pp. 180-194, 2022.
3. A. Kampmann, A. Mokhtarian, S. Kowalewski, and B. Alrifaee, "ASOA - A dynamic software architecture for software-defined vehicles," in Proc. Aachen Colloquium Sustainable Mobility, 2022, pp. 1-7.
4. F. Pan, J. Lin, M. Rickert, and A. Knoll, "Resource allocation in software-defined vehicles: ILP model formulation and solver evaluation," in Proc. IEEE 25th Int. Conf. Intelligent Transportation Systems (ITSC), 2022, pp. 2577-2584.
5. L. Wen, M. Rickert, F. Pan, J. Lin, and A. Knoll, "Bare-metal vs. hypervisors and containers: Performance evaluation of virtualization technologies for software-defined vehicles," in Proc. IEEE Intelligent Vehicles Symposium (IV), 2023, pp. 1-8.
6. G. Q. Xie, W. Wu, G. Zeng, R. F. Li, and S. Y. Hu, "Risk assessment and development cost optimization in software defined vehicles," IEEE Transactions on Intelligent Transportation Systems, vol. 24, no. 7, pp. 7375-7386, 2023.
7. International Organization for Standardization, ISO 24089:2023, Road vehicles - Software update engineering, 2023.
8. International Organization for Standardization, ISO/SAE 21434:2021, Road vehicles - Cybersecurity engineering, 2021.
9. United Nations Economic Commission for Europe, UN Regulation No. 155 - Cyber security and cyber security management system, 2021.
10. United Nations Economic Commission for Europe, UN Regulation No. 156 - Software update and software update management system, 2021.
11. National Institute of Standards and Technology, Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities, NIST SP 800-218, 2022.
12. National Institute of Standards and Technology, Artificial Intelligence Risk Management Framework (AI RMF 1.0), NIST AI 100-1, 2023.
13. National Institute of Standards and Technology, The NIST Cybersecurity Framework (CSF) 2.0, NIST CSWP 29, 2024.
14. R. David et al., "TensorFlow Lite Micro: Embedded machine learning on tinyML systems," in Proc.

MLSys, 2021.
15. C. Banbury et al., "MLPerf Tiny Benchmark," in Proc. NeurIPS Datasets and Benchmarks Track, 2021.
16. H. Ren, D. Anicic, and T. A. Runkler, "TinyOL: TinyML with online-learning on microcontrollers," in Proc. Int. Joint Conf. Neural Networks (IJCNN), 2021.
17. G. Demosthenous and V. Vassiliades, "Continual learning on the edge with TensorFlow Lite," arXiv:2105.01946, 2021.
18. M. N.-U.-R. Chowdhury, A. Haque, H. Soliman, M. S. Hossen, T. Fatima, and I. Ahmed, "Android malware detection using machine learning: A review," arXiv:2307.02412, 2023.
19. Andrid Developers Blog, "Build smarter Android apps with on-device Machine Learning," Google, 2023.
20. Android Open Source Project, "Android Automotive OS power management," Google, 2023.
21. OWASP Foundation, Mobile Application Security Verification Standard (MASVS) 2.0, 2023.
22. S. Rose, O. Borchert, S. Mitchell, and S. Connelly, Zero Trust Architecture, NIST SP 800-207, 2020.
23. E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.3, RFC 8446, Internet Engineering Task Force, 2018.
24. T. K. Kuppusamy, L. A. DeLong, and J. Cappos, "Securing software updates for automobiles," in Proc. ESCAR USA, 2016.
25. T. K. Kuppusamy, L. A. DeLong, and J. Cappos, "Securing software updates for automotives using Uptane," USENIX ;login:, vol. 42, no. 2, 2017.