A Tutorial on Secure and Efficient Firmware Delivery

Dakshina Tharindu, Aruna Jayasena, and Prabhat Mishra
University of Florida, USA

***Abstract*—This tutorial analyzes various aspects of firmware delivery and how they are affected by the introduction of post-quantum safe cryptographic algorithms. Specifically, we describe various firmware delivery stages from both the manufacturer (packaging) and device (deployment) perspectives, including compression (decompression), encryption (decryption), and signing (verification). Extensive experimental evaluation using both real and synthetic benchmarks explores the trade-off between firmware delivery effort (time) and security requirements. Interestingly, our results show that adopting advanced post-quantum digital signature algorithms does not significantly increase firmware deployment time, highlighting their practical viability in next-generation secure firmware delivery pipelines.**

***Index Terms*—Firmware delivery, firmware security**

I. Introduction

Modern computing devices depend on the unified integration of hardware, firmware, and software to deliver their intended functionality. Firmware plays an important role in these systems, managing and coordinating the operation of hardware components while serving as an interface to higher-level software applications. Figure 1 shows an illustrative example of an automotive system, where various embedded control systems perform independent tasks, such as cruise control, airbag control, and engine control, while working in harmony to control the vehicle. For example, the adaptive cruise control system collects data from various sensors, such as radar, cameras, and ultrasonic devices, and processes it with advanced application software. The decisions made by the application software are then relayed to actuators, such as the drive-by-wire system, to maintain an appropriate distance from the vehicle ahead. Firmware coordinates this process by managing sensor inputs, initializing applications, and ensuring accurate actuator control. Similarly, the firmware in the other control systems works as a bridge between the hardware components and the operating system.

When functional or security issues are discovered in the hardware components after deployment, firmware updates are often required to address these concerns. In the context of vehicles, such updates may be performed during a dealer visit or through over-the-air (OTA) updates [1]. This raises a critical challenge: how can the device (e.g., car) verify that the firmware update originates from a trusted source and ensure that the update has not been compromised by malicious implants. In this paper, we discuss the firmware delivery process and analyze it in terms of security levels as well as the time and effort involved when implementing various cryptographic algorithms.

This tutorial provides an overview of secure firmware delivery, covering confidentiality, authentication, packaging,

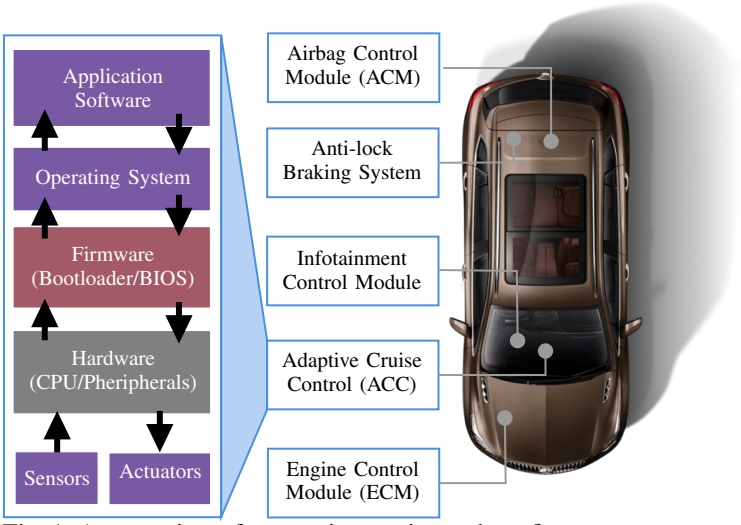

Fig. 1: An overview of system interactions where firmware acts as a bridge between the hardware and the operating system.

and deployment from both the manufacturer and device perspectives. Unlike existing surveys or vendor documents that treat these elements separately, this work unifies the complete firmware delivery pipeline and examines how post-quantum cryptographic (PQC) algorithms affect real-world update workflows. We systematically examine the roles of compression, encryption, and digital signatures in preserving security properties, and analyze how these mechanisms interact to influence performance. The objective is to provide practical guidance for selecting cryptographic and compression strategies based on application-specific security and performance constraints, and to assess the feasibility of transitioning toward PQC-ready firmware infrastructures.

This tutorial is organized as follows. We first provide background and survey related efforts. Next, we discuss confidentiality, authentication, and firmware packaging for delivery. Finally, we present experimental results to compare firmware delivery time for different configurations.

II. Background and Related Work

Firmware ensures that devices operate securely and correctly, with updates providing functional enhancements, bug fixes, and security patches [2]–[4]. Secure firmware deployment must satisfy several key requirements:

- *Authenticity*: Public key infrastructure (PKI) is used to sign and verify firmware to ensure authenticity.
- *Confidentiality*: Encryption preserves the confidentiality of the firmware binary.

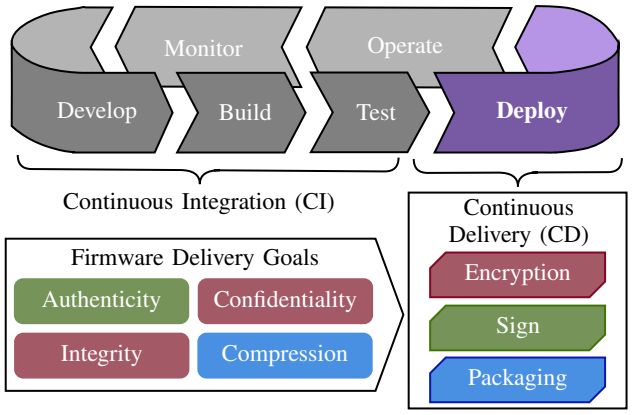

Fig. 2: Different stages of continuous integration and continuous delivery (CICD) pipeline.

- *Fast Delivery*: Compression reduces firmware size, minimizing network traffic and improving delivery efficiency.
- *Anti-Rollback*: Anti-rollback mechanisms prevent installation of older firmware versions, even if signed.

Figure 2 shows the continuous integration and continuous delivery (CICD) pipeline for firmware. While similar to software pipelines, firmware delivery has notable differences: (1) firmware manages its own update process rather than relying on an operating system; (2) updates must tolerate failures without bricking the device; and (3) new firmware must authenticate itself to the currently running firmware.

The early pipeline stages include development, build, and testing, during which manufacturers validate functional and non-functional requirements. Development typically involves low-level languages such as C or assembly optimized for target hardware. Extensive testing and validation aim to ensure correctness and minimize vulnerabilities. After testing, the firmware is deployed to devices in the field through a process consisting of the following steps:

Manufacturer Viewpoint (Provisioning): Once developed, firmware must be securely distributed to end-users. Manufacturers package firmware binaries with metadata and cryptographic verification data, such as hashes or digital signatures. Distribution is performed via over-the-air updates, physical media, or secure download portals, using encryption and PKI to prevent tampering and unauthorized access.

Device Viewpoint (Deployment): Deployment installs or updates firmware using mechanisms that ensure hardware compatibility and security. Devices verify cryptographic signatures before installation and may employ fail-safes such as dual-partition systems to support rollback. Updates can be applied manually or automatically across large device fleets.

A. Related Work

Uptane is the de facto industry standard for secure over-the-air (OTA) firmware updates in modern vehicles [5]. It employs signed metadata roles, role separation, and rollback protection to ensure firmware integrity under adversarial conditions. For IoT and embedded systems, the ARM Platform Security Architecture (PSA) defines guidelines for secure firmware update workflows, including trusted boot, secure storage, key management, and anti-rollback enforcement. Trusted Firmware-M implements PSA principles on ARM Cortex-M platforms, providing a reference architecture for secure boot, cryptographic services, metadata validation, and protected firmware updates in resource-constrained environments.

Kolehmainen [6] presents a survey of secure firmware update mechanisms that categorize existing approaches and highlight their suitability for resource-constrained IoT devices. Choi [7] proposes a secure firmware validation and update framework for consumer devices in home networking environments. Nilsson [8] investigates secure over-the-air firmware updates, addressing challenges such as high mobility and strict safety requirements. Lee [9] introduces a blockchain-based firmware update mechanism for IoT systems. Zandberg [10] provides a practical evaluation of open standards for secure firmware updates in constrained IoT devices. Wu [11] examines vulnerabilities across multiple firmware update platforms, exposing systemic weaknesses that attackers can exploit. While existing surveys focus on a specific aspect of firmware delivery, this tutorial provides a comprehensive overview of firmware delivery from both manufacturer (packaging) and device (deployment) perspectives.

III. Preserving Firmware Confidentiality

Firmware confidentiality prevents unauthorized access, reverse engineering, and intellectual property theft, particularly when firmware contains proprietary logic, credentials, or operates in untrusted environments. Regulatory standards in automotive, medical, and defense systems often require strong confidentiality guarantees. Encryption protects firmware throughout its lifecycle (storage, transmission, and installation), mitigating risks such as cloning, counterfeit devices, compromised update channels, and local extraction attacks. Below, we outline a secure procedure for preserving firmware confidentiality from manufacturer to device.

A. Manufacturer Perspective (Provisioning)

Manufacturers encrypt firmware prior to distribution to ensure that only authorized devices can access it. This process consists of the following steps.

Compress Prior to Encryption: Firmware is compressed using lossless techniques to reduce size and improve transmission efficiency. Compression must precede encryption, as encrypted data is not compressible.

Select an Encryption Algorithm: Manufacturers select secure encryption algorithms, typically symmetric schemes such as the Advanced Encryption Standard (AES) at 128- or 256-bit key lengths, or ChaCha20, for efficiency with large binaries. Asymmetric algorithms such as RSA (Rivest–Shamir–Adleman) and Elliptic Curve Cryptography (ECC) are less commonly used for bulk firmware encryption.

Generate and Manage Encryption Keys: For symmetric encryption, unique keys are generated per firmware release and securely shared with authorized devices. Asymmetric schemes use the device's public key for encryption, ensuring only the corresponding private key can decrypt.

Encrypt the Firmware: The firmware binary is encrypted using the algorithm and key. Required metadata, such as the algorithm identifier, version, and keying information, accompanies the encrypted firmware to support device-side decryption.

B. Device Perspective (Deployment)

The device is responsible for securely decrypting the received firmware package and constructing the binary version of the firmware update. It consists of three major steps.

Extract Metadata and Encrypted Firmware: The device extracts encryption metadata and the encrypted firmware binary, configuring parameters such as encryption algorithm, initialization vectors, and related parameters.

Retrieve Decryption Keys: The device retrieves the decryption key. Symmetric keys are typically stored in a Trusted Execution Environment (TEE) or Hardware Security Module (HSM), while asymmetric schemes use the device's private key.

Decrypt the Firmware: The device decrypts the firmware using the specified algorithm and key. Decryption failures, caused by tampering or incorrect keys, result in the update being rejected.

Ensuring firmware confidentiality requires more than choosing an encryption algorithm. Manufacturers must address key provisioning, including secure key injection during production and protection of these keys throughout the device's lifetime.

IV. Firmware Authentication Procedure

Authentication is critical in firmware updates, enabling devices to verify the firmware publisher's identity through digital signatures using asymmetric cryptography, where the device uses a trusted public key to verify signatures generated with the vendor's private key. This process ensures firmware authenticity and integrity, preventing attackers from introducing malicious code with backdoors or vulnerabilities that could compromise device security. Without proper authentication, devices become vulnerable to man-in-the-middle attacks, unauthorized firmware installations, and tampering during distribution. This protection is especially vital for sensitive applications in healthcare, financial, automotive, and industrial systems, where verified digital signatures and cryptographic hashes establish trust, ensure compliance with security standards, and reduce risks of operational disruptions, data breaches, or device hijacking.

A. Manufacturer Perspective (Provisioning)

Figure 3 illustrates an overview of the process involved in the firmware authentication from the manufacturer's side. In this section, we examine each of these stages that facilitate secure delivery by the manufacturer in detail.

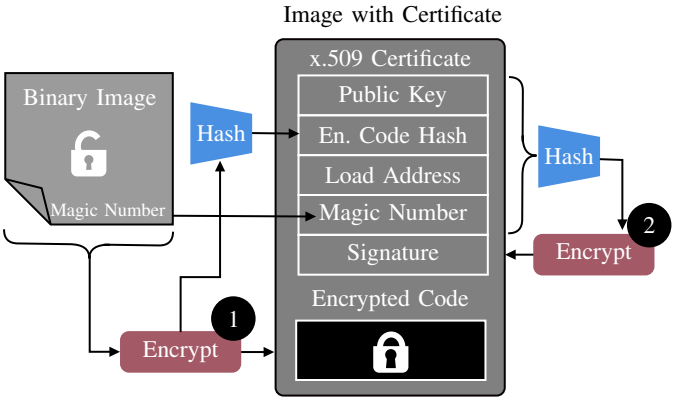

Fig. 3: Illustration showing the sequence of actions taken by the manufacturer during the firmware authentication step

Generation of Asymmetric Key Pair: Manufacturers select suitable asymmetric algorithms based on device constraints and security requirements. Common choices include RSA (2048-4096 bits) or ECC algorithms (ECC-224 to ECC-521), with ECC preferred for resource-constrained devices due to shorter keys and lower computational overhead. The generated key pair consists of a private key securely stored by the manufacturer and a public key embedded in the device during manufacturing for firmware verification.

Compute the Hash of Firmware Update: The manufacturer generates a cryptographic hash of the firmware binary (or encrypted binary) to create a fixed-length data representation. This ensures data integrity and reduces signing complexity. Common hash algorithms include SHA-256, SHA-384, SHA-512, or SHA-3 family algorithms, chosen based on security requirements and signing algorithm compatibility.

Sign the Hash and Attach to Firmware: The computed hash is signed using the manufacturer's private key, creating a digital signature. The signature is appended to the firmware binary as part of the update package, often with additional metadata like version numbers, timestamps, or algorithm identifiers for proper update tracking and compatibility.

B. Device Perspective (Deployment)

Figure 4 illustrates the checks a device performs before installing a firmware update. This section outlines the steps taken by the device to verify the authenticity and integrity of the received firmware update package.

Extract the Digital Signature and Metadata: The device begins by extracting the signature and any associated metadata (e.g., version number, hash algorithm identifier, etc.) from the firmware update package. The extracted signature represents the manufacturer's cryptographic proof of authenticity.

Compute the Hash: The device computes a cryptographic hash of the received firmware binary using the same hash algorithm specified in the metadata or preconfigured during the device's initial setup. This ensures consistency with the hash originally generated by the manufacturer.

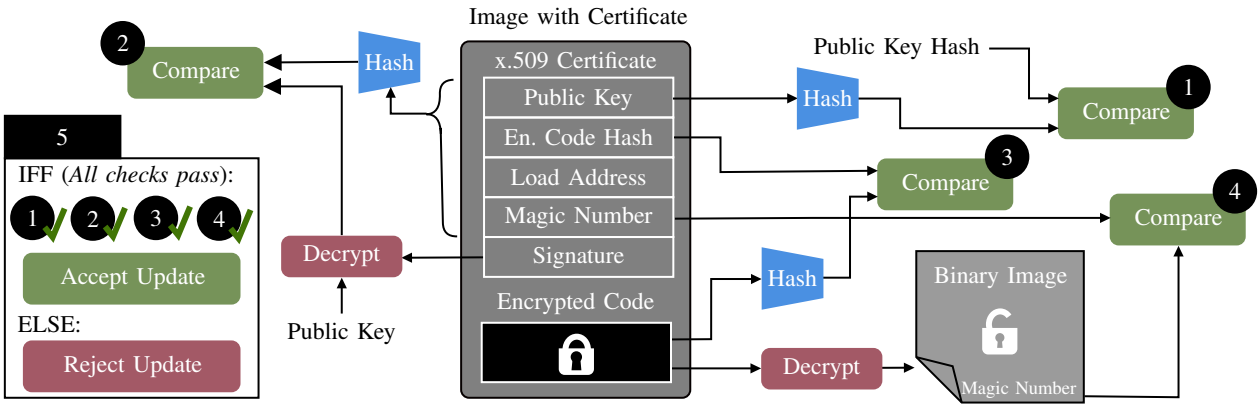

Fig. 4: Illustration showing the sequence in which system firmware applies when authenticating a firmware update.

Verify the Signature using the Public Key: Using the embedded public key, the device verifies the digital signature by decrypting it to retrieve the original hash computed by the manufacturer. This step ensures that the firmware update was signed by a trusted source and has not been tampered with.

Compare the Hashes: The device compares the hash it computed from the received firmware with the hash obtained from verifying the digital signature.

- If the two hashes match, the firmware update is confirmed to be authentic and untampered. The device proceeds with installing the update.
- If the hashes do not match, the firmware update is rejected to prevent malicious or corrupted updates.

This process ensures that only trusted firmware updates, verified by cryptographic mechanisms, are installed on the device, protecting it against tampering and unauthorized changes.

V. Firmware Packaging for Delivery

Once encryption and authentication are done, the firmware needs to be sent to the end-user (device) for deployment. This can be performed in two ways: tethered updates or over-the-air updates. The encrypted firmware binary and its associated metadata are securely bundled into one package. A well-structured package not only safeguards the firmware against tampering and unauthorized access but also provides the necessary information for authentication, decryption, and installation. We outline the steps in creating a firmware package from both the manufacturer and device perspectives.

A. Manufacturer Perspective (Provisioning)

The manufacturer is responsible for constructing the firmware package that includes all essential components while maintaining security and compatibility with the device.

Include the Encrypted Firmware: The core component is the encrypted firmware binary, ensuring confidentiality during transmission as discussed in Section . III.

Attach Authentication Certificate and Metadata: The authentication signature and related components are attached to the package. Metadata includes version information (preventing downgrade attacks), hash algorithm details, encryption algorithm specifications, and device compatibility information.

Generate the Package: The firmware is packaged into a secure container format with additional elements like checksums, versioning information, and integrity headers. These components enable devices to verify completeness and authenticity upon receipt, detecting issues like corrupted or incomplete downloads. The resulting container serves as the final distributable firmware image ready for deployment.

B. Device Perspective (Deployment)

Once the packaged firmware reaches the device, it needs to be properly decoded and prepared for necessary actions such as authentication, decryption, and installation.

Validate Package Integrity: Before extracting the package, the device verifies its integrity using the checksum or other redundancy mechanisms in the container format. It ensures that the package was not corrupted during transmission.

Unpack the Components: Once the initial validation is completed, the received image is unpacked to obtain the firmware binary and metadata from the package. Metadata is decoded into respective components to select the cryptographic algorithms and other parameters. If the image is compressed, it needs to be uncompressed after the decryption process.

Anti-Rollback Version Check: The device performs an anti-rollback version check to prevent the installation of outdated or potentially vulnerable firmware versions. Using the version number embedded in the metadata, the device compares it against a securely stored version counter. If the received firmware version is older or equal to the current version, the update is rejected to safeguard against downgrade attacks.

Process Security Components: The device retrieves the cryptographic signature, hash algorithm identifier, and encryption details from the metadata to perform decryption and authentication, using the steps outlined in Section III and Section IV.

Prepare for Installation: After verifying firmware integrity and authenticity, devices are prepared for installation using techniques like staging or dual-system partitions to ensure reliable updates and minimize downtime. Staging partitions

provide a temporary area where updates are applied and validated before overwriting active firmware, preventing failures from corruption or interruptions.

Real-world update systems rely on structured container formats and delivery infrastructures. For example, automotive ECUs commonly use Uptane-compliant metadata and signed binaries, while embedded microcontrollers employ formats such as MCUboot images with TLV-encoded metadata or Intel's Firmware Interface Table (FIT) in larger SoCs. Similarly, mobile platforms, such as Android, use A/B partitioning with compressed OTA packages that include signatures, payload metadata, and delta updates.

VI. Experiments

In this section, we first examine a real-world example of firmware delivery. Next, we analyze various firmware delivery configurations to evaluate their effectiveness in terms of packaging time, deployment time, security, and compression ratio, which impact network usage during the firmware update.

A. Case Study: Firmware Update in Vehicles

Modern vehicles incorporate 50-150 ECUs controlling safety-critical functions, including braking, engine control, and assisted driving. Firmware updates traverse a constrained and security-critical delivery chain. The firmware image is compressed to meet OTA bandwidth limits, encrypted using symmetric keys provisioned at production, and signed with a long-term OEM private key. Metadata describing ECU compatibility, versioning, rollback constraints, and certificate chains is bundled into a secure update container. On the vehicle, the telematics unit receives the update and verifies it using certificates stored in the hardware root of trust. If validation succeeds, the update is staged in a dual-bank flash layout, enabling recovery from interruptions such as power loss or network failure. After installation, the ECU switches execution to the new firmware and updates rollback counters to prevent downgrade attacks.

B. Experimental Setup

We used the Zephyr real-time operating system (RTOS) kernel and a Mini neural network application. The experiments were carried out using ARM Cortex-A9 processors. All experiments were conducted on a PC equipped with an Intel Core i7-9750H CPU @ 2.60GHz and 16GB of RAM. QEMU was used to execute the final firmware in the target environment and measure the deployment times taken by different firmware configurations.

We have explored various firmware delivery configurations consisting of different encryption algorithms, authentication methods, and compression techniques. Specifically, we used the Lempel-Ziv-Welch (LZW*) and Lempel–Ziv–Markov chain algorithm 2 (LZMA2) compression techniques. For LZW*, we used the *compress* tool and, for LZMA2, we used the *xz* tool. Note that LZW* (*compress*) is an adaptive LZW that utilizes an effective combination of LZW and Huffman coding. We used the *OpenSSL* library to implement authentication and encryption algorithms. For authentication, we used a variety of algorithms, including: RSA 3072, RSA 4096, ECDSA 192 (Elliptic Curve Digital Signature Algorithm), ECDSA 384, and two post-quantum algorithms, Dilithium 20224 and Falcon 13056. We used the SHA-3 512 as the hash algorithm during all the experiments. For implementing the post-quantum authentication, we used the *oqs-provider* [12], an extension to the standard *OpenSSL* library. For encryption, we used AES 128 CBC (Cipher Block Chaining), AES 256 CBC, AES 128 CTR (Counter Mode), AES 256 CTR, and ChaCha20_Poly1305 algorithms.

C. Firmware Delivery Time for Various Configurations

Table I compares firmware delivery time (in ms) for various configurations shown in the first column (C_1 - C_{16}). The next three columns show different authentication, encryption, and compression methods, respectively. The remaining columns indicate Compression Ratio (CR), Sign Time (ST), Encryption Time (ET), Compression Time (CT), Packaging Time (PT), Verify Time (VT), Decryption Time (DecT), Decompression Time (DcmpT), and Deployment Time (DT). Note that packaging time (PT) $= ST + ET + CT$ and deployment time (DT) $= VT + DecT + DcmpT$.

The results in Table I demonstrate significant variations in packaging time (PT) and deployment time (DT) across different algorithm combinations, revealing important trade-offs between security strength and performance. Examining packaging time, configurations using LZMA2 compression consistently achieve superior compression ratios (0.66 vs. 0.98 for LZW*) but incur substantially lower packaging times. For instance, comparing C_1 (LZW*) with C_2 (LZMA2), both using identical authentication and encryption schemes, the LZMA2 configuration achieves $PT = 46.46$ ms compared to $47.76ms$ for LZW*, a modest difference attributable to the dramatically reduced compression time (29.15 ms vs. 30.14 ms). However, the compression efficiency advantage becomes more pronounced with larger key sizes: C_3 using AES-256-CBC with LZW* exhibits $PT = 384.99ms$, whereas C_4 with LZMA2 achieves only $47.85ms$. The choice of authentication algorithm also significantly impacts timing, with post-quantum algorithms generally imposing higher overheads. Falcon-based configurations (C_{13} through C_{16}) demonstrate packaging times ranging from $50.41ms$ to $431.09ms$, depending on compression choice.

Deployment time follows similar patterns, where LZMA2 configurations consistently outperform LZW* variants: C_{10} achieves $DT = 29.42ms$ compared to C_9's $98.402ms$. Notably, configurations employing ChaCha20 encryption show competitive performance, with C_{15} achieving $DT = 37.51ms$ despite using LZW* compression.

We can make two important observations from these findings. First, compression and decompression dominate both packaging time and deployment time across configurations. Therefore, optimizing compression may yield higher performance gains than changing cryptographic primitives. Sec-

TABLE I: Time (ms) taken by each of the firmware delivery components for ARM Cortex A9 development board with the Mini neural network application running on Zephyr real-time operating system with a communication thread. We used the following notations: Compression Ratio (CR), Sign Time (ST), Encryption Time (ET), Compression Time (CT), Packaging Time (PT), Verify Time (VT), Decryption Time (DecT), Decompression Time (DcmpT), and Deployment Time (DT). Note that packaging time (PT) $= ST + ET + CT$ and deployment time (DT) $= VT + DecT + DcmpT$

Config	*Authentication*	*Encryption*	*Compression*	*CR*	*ST*	*ET*	*CT*	*PT*	*VT*	*DecT*	*DcmpT*	*DT*
C1	RSA 3072	AES-128-CBC	LZW*	0.98	10.38	7.23	30.14	47.76	7.71	5.79	18.61	32.12
C2	RSA 3072	AES-128-CBC	LZMA2	0.66	10.31	6.99	29.15	46.46	7.68	5.68	17.18	30.55
C3	RSA 3072	AES-256-CBC	LZW*	0.98	8.82	5.78	370.38	384.99	6.27	5.04	87.41	98.73
C4	RSA 3072	AES-256-CBC	LZMA2	0.66	10.74	7.66	29.44	47.85	7.84	5.79	17.38	31.01
C5	ECDSA 192	AES-256-CBC	LZW*	0.66	6.72	6.26	370.36	385.31	6.76	5.12	86.51	97.92
C6	ECDSA 192	AES-128-CBC	LZMA2	0.98	8.23	6.97	29.15	44.36	8.20	5.58	17.11	30.89
C7	ECDSA 192	AES-256-CBC	LZW*	0.66	6.69	5.79	368.48	380.96	6.76	4.91	85.90	97.58
C8	ECDSA 192	AES-256-CBC	LZMA2	0.98	8.23	7.79	29.92	45.94	8.26	5.96	18.26	32.49
C9	Dilithium	AES-128-CTR	LZW*	0.66	6.72	6.265	370.368	383.353	6.767	5.124	86.511	98.402
C10	Dilithium	AES-256-CTR	LZMA2	0.98	7.02	5.97	29.06	42.04	6.76	5.61	17.05	29.42
C11	Dilithium	AES-128-CTR	LZW*	0.66	5.42	5.23	371.85	382.50	5.32	5.12	87.21	97.65
C12	Dilithium	AES-256-CTR	LZMA2	0.98	6.93	6.23	29.40	42.56	6.81	5.80	17.59	30.20
C13	Falcon	AES-256-CTR	LZW*	0.98	8.73	7.76	33.92	50.41	8.67	8.75	20.60	38.02
C14	Falcon	AES-256-CTR	LZMA2	0.66	7.24	6.73	411.70	425.67	6.93	7.21	89.52	103.66
C15	Falcon	ChaCha20	LZW*	0.98	8.95	10.16	31.44	50.55	8.41	10.33	18.77	37.51
C16	Falcon	ChaCha20	LZMA2	0.66	7.32	7.37	416.40	431.09	6.73	7.67	91.90	106.30

ond, the results show that post-quantum signature schemes (Dilithium and Falcon) introduce acceptable overhead. Moreover, verification time remains small relative to decompression time, suggesting that integrating PQC authentication into firmware delivery pipelines is practical. Overall, firmware vendors must balance compression efficiency, security strength, and computational overhead based on application-specific constraints. These findings underscore that firmware vendors must carefully select appropriate combinations of authentication, encryption, and compression techniques aligned with their intended security level and performance requirements.

VII. Conclusion

In this paper, we performed a comprehensive analysis of the firmware delivery process. Specifically, we discussed the significant changes in the firmware delivery pipeline from the regular software delivery steps. Next, we discussed the three important stages in the firmware delivery process, including authentication, confidentiality, and deployment, from both the manufacturer (packaging) and the end-user (device) perspectives. Our analysis revealed that a vendor needs to consider the trade-off between security strength, compression efficiency, computational overhead, and deployment latency to match application-specific firmware delivery constraints.

References

[1] Russ Bielawski, Ron Gaynier, et al. Cybersecurity of firmware updates. Technical report, National Highway Traffic Safety Administration, U.S. Department of Transportation, 2020.

[2] Aruna Jayasena and Prabhat Mishra. Firmwall: Directed symbolic execution of firmware binaries for defending against unauthorized system calls. *IEEE Transactions on Information Forensics and Security*, 20:6999–7012, 2025.

[3] Aruna Jayasena and Prabhat Mishra. Hive: Scalable hardware-firmware co-verification using scenario-based decomposition and automated hint extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(10):3278–3291, 2024.

[4] Dakshina Tharindu, Aruna Jayasena, and Prabhat Mishra. Sysfuss: System-level firmware fuzzing with selective symbolic execution. *arXiv preprint arXiv:2602.02243*, 2026.

[5] Trishank Karthik, Akan Brown, Sebastien Awwad, et al. Uptane: Securing software updates for automobiles. In *International conference on embedded security in car*, volume 11, 2016.

[6] Antti Kolehmainen. Secure firmware updates for iot: A survey. In *IEEE International Conference on Internet of Things*, pages 112–117, 2018.

[7] Byung-Chul Choi, Seoung-Hyeon Lee, Jung-Chan Na, and Jong-Hyouk Lee. Secure firmware validation and update for consumer devices in home networking. *IEEE Transactions on Consumer Electronics*, 62(1):39–44, 2016.

[8] Dennis K Nilsson and Ulf E Larson. Secure firmware updates over the air in intelligent vehicles. In *IEEE International Conference on Communications Workshops*, pages 380–384, 2008.

[9] Boohyung Lee and Jong-Hyouk Lee. Blockchain-based secure firmware update for embedded devices in an internet of things environment. *The Journal of Supercomputing*, 73:1152–1167, 2017.

[10] Koen Zandberg, Kaspar Schleiser, Francisco Acosta, Hannes Tschofenig, and Emmanuel Baccelli. Secure firmware updates for constrained iot devices using open standards: A reality check. *IEEE access*, 7:71907–71920, 2019.

[11] Yuhao Wu, Jinwen Wang, Yujie Wang, Shixuan Zhai, Zihan Li, Yi He, Kun Sun, Qi Li, and Ning Zhang. Your firmware has arrived: A study of firmware update vulnerabilities. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5627–5644, 2024.

[12] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In *International Conference on Selected Areas in Cryptography*, pages 14–37. Springer, 2016.

Dakshina Thariundu is a Ph.D. student in the Department of Computer and Information Science and Engineering at the University of Florida. His research is focused on hardware security and trustworthy AI.

Aruna Jayasena is an Assistant Professor at the University of Tennessee - Chattanooga. He received his Ph.D. in the Department of Computer and Information Science and Engineering at the University of Florida in 2025. His research focuses on heterogeneous system design, applied cryptography, trusted execution, and hardware-firmware co-validation.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include embedded and cyber-physical systems, hardware security and trust, and energy-aware computing. He is an IEEE Fellow, an AAAS Fellow, and an ACM Distinguished Scientist.