Securing firmware updates using Transparency and Traceability services

Nikos Fotiou,* Lefteris Georgiadis,* George C. Polyzos,*†‡ Vasilios A. Siris*†
* ExcID, Athens, Greece
† Mobile Multimedia Laboratory, Department of Informatics,
School of Information Sciences and Technology, Athens University of Economics and Business, Greece
‡ School of Data Science, Chinese University of Hong Kong, Shenzhen, China

***Abstract*—Firmware update mechanisms are a critical part of the software supply chain in IoT and embedded systems, yet they remain vulnerable to a range of attacks—particularly those targeting the signing keys of firmware authors. A compromised signing key can be used to distribute malicious firmware that appears legitimate, undermining the integrity of the entire update process. In this paper, we address this class of attacks by introducing transparency and traceability services into the firmware update pipeline. We propose the use of a *transparency registry*—an immutable, append-only log where all firmware signing events are recorded. By monitoring this registry, key compromise and unauthorized signing activity can be detected and mitigated. Our approach shifts trust from opaque signature validation to verifiable, auditable records of signing activity. To evaluate this approach, we design and implement two systems: one based on a centralized transparency registry built using Merkle tree structures, and another based on a decentralized, permissioned blockchain. Both implementations demonstrate that transparency-based defenses against firmware supply chain attacks are practical, scalable, and effective.**

***Index Terms*—key transparency, blockchain, supply chain**

I. Introduction

Firmware is a critical component of modern digital infrastructure, embedded in devices ranging from consumer electronics to industrial control systems. Its foundational role and privileged access make it a high-value target for attackers, yet traditional protections–such as digital signatures–are increasingly inadequate in the face of evolving threats. While digital signatures provide a cryptographic assertion of authenticity, they do not inherently guarantee transparency, integrity of the signing process, or protection against key compromise and insider threats.

Recent attacks on software supply chains have demonstrated that merely verifying the origin of firmware is insufficient. If the signing key is stolen or misused, malicious firmware can appear just as legitimate trusted releases. To address this shortfall, transparency frameworks offer a powerful complementary feature: by recording signing events in a tamper-evident, append-only log, firmware producers can create an auditable trail of signing activity. This enables independent verification of whether a firmware image was genuinely approved and signed through legitimate channels.

In this paper, we focus on securing firmware updates by integrating transparency into the firmware signing process. We propose the use of a transparency registry–a verifiable log structure that records signing events–to elevate trust from merely accepting a digital signature to verifying the context and legitimacy of the signing operation itself. By committing each signing action to a public log, we make the process observable, allowing firmware consumers, security monitors, and automated systems to detect anomalies, enforce policies, and ultimately reduce the risk of undetected compromise. This approach does not replace digital signatures, but strengthens their trustworthiness by embedding them in a broader framework of transparency and traceability.

The remainder of this paper is organized as follows. We present the design of our solution in Section II. We introduce our approaches for implementing transparency registries in Section III. We present our implementation in Section IV. Finally, we discuss related work in Section V and we conclude our paper in Section VI.

II. Design

A. Entities and interactions

Our architecture builds upon the entities and interactions defined in RFC 9019 [1] (see also Fig. 1). In our architecture an *Author* is responsible for creating a firmware *image file*. The Author produces both the firmware binary and an associated *manifest file* containing necessary metadata such as version information, hash values, and digital signatures. Once a firmware image is prepared, the Author makes it available through a *firmware server* managed by the *Device Operator*. A Device Operator is responsible for managing IoT devices: IoT device owners and end-users interact with their devices indirectly—typically through device operators, e.g., using a Web portals or a smartphone application.

A Device Operator operates a component called the *status tracker* that continuously monitors for new firmware versions and tracks the update status of enrolled IoT devices. When a new firmware version is available, IoT devices are notified via the *status tracker client*–a software component running on each device. This notification can be implemented using either a pull-based approach (where devices periodically check for updates) or a push-based approach (where updates are actively signaled by the Device Operator).

After receiving the notification, a *firmware consumer* component on the IoT device connects to the firmware server, downloads the new firmware binary file, and retrieves the

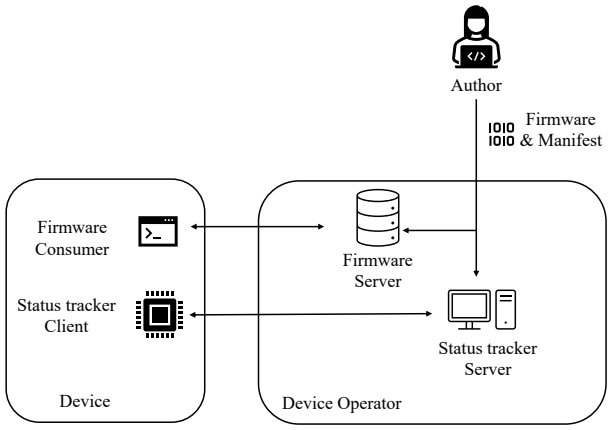

Fig. 1. A firwmare update architecture based on RFC 9019.

corresponding manifest file. The firmware consumer is responsible for validating the firmware image before initiating the update process.

B. Trust model

We frame the firmware update architecture described in the previous section using the Claimant Model,[1] which defines a trust model involving the following roles:

- **Claimant:** An entity that asserts a claim by issuing a signed *statement*.
- **Believer:** An entity that relies on the statement to believe a claim.
- **Verifier:** An entity responsible for independently verifying the accuracy and legitimacy of claims (through the statements).
- **Arbiter:** An entity that resolves disputes when false or conflicting claims are detected. This could be a human auditor, a security monitoring service, or an automated policy enforcement system that is notified when a claim is proven to be false.

This model provides a structured way to reason about the authenticity and integrity of firmware updates. The security of this trust model relies on the assumption that Verifiers have timely access to the same signed statements as Believers and are able to verify the corresponding claims. If a Verifier detects a false or conflicting claim, it can alert a Arbiter to take appropriate action.

As an example, based on the architecture described in the previous subsection, consider the claim $C1$: *"The digest of firmware file f is d."* With respect to this claim, the roles of our trust model are assigned to the following entities:

With respect to this claim, it can be easily observed that our architecture is secure, since the Verifier and the Believer can be the same entity (i.e., the *firmware consumer* component on the IoT device).

[1]https://github.com/google/trillian/blob/master/docs/claimantmodel/CoreModel.md

Claimant	The firmware *Author* who produces the firmware image and generates a corresponding signed manifest file. This manifest file constitutes the statement supporting the claim, and its authenticity can be verified using the Author's public key.
Believer	The *firmware consumer* component on the IoT device, which relies on the signed manifest to determine the validity and trustworthiness of the firmware file. The *firmware consumer* receives the signed file from the *firmware server*.
Verifier	Any of the following: *firmware consumer* component on the IoT device, *firmware server* performing internal checks, or trusted third-party verification service.
Arbiter	The *firmware Author*, which may revoke invalid firmware, and trusted third-party security solution providers.

TABLE I
CLAIMANT MODEL FOR CLAIM $C1$.

Let us now consider another critical claim essential for securing the IoT firmware update process, claim $C2$: *"The file f is a valid firmware for device D."*

Here, *valid* means that the firmware file has indeed been created and approved by the *Author*, and is intended for installation on device D. With respect to this claim, the roles of our trust model are assigned as follows:

Claimant	The firmware *Author*, who produces the firmware image and generates a corresponding signed manifest file asserting its validity.
Believer	The *firmware consumer* component on the IoT device, which relies on the signed manifest to assess whether the firmware file is trustworthy and appropriate for device D. The firmware and manifest are retrieved from the *firmware server*
Verifier	Only the firmware *Author* can act as a Verifier in this case. The Author periodically downloads the firmware and its signed manifest from the *firmware server* and checks their correctness to ensure no tampering has occurred.
Arbiter	The *firmware Author*, who may revoke a compromised firmware version, and trusted third-party security solution providers that may monitor or intervene upon detection of a violation.

TABLE II
CLAIMANT MODEL FOR CLAIM $C2$.

With respect to this claim, the following attacks highlight gaps in the considered architecture:

1) Key breach, fake firwmare server: A malicious entity compromises the Author's signing key, creates and signs a malicious firmware version, then deploys it on a fake firmware server. The *firmware consumer* is tricked into downloading and installing the malicious version from the unauthorized server.

2) Key breach, firwmare server compromised: A malicious entity gains access to both the Author's signing key and the legitimate firmware server. It signs and uploads a malicious firmware version, temporarily replacing the valid one. The attack remains undetected between the periodic verification intervals of the Author.

3) Key breach, firwmare server compromised, split view: A more advanced version of Attack 2. The malicious actor again controls both the signing key and the firmware server.

However, in this case, the server selectively serves the valid firmware only when queried by the Author, while serving the malicious version to the *firmware consumer*. This bypasses detection by providing a *split view* of firmware.

These attacks succeed because the *Verifier* does not have guaranteed, real-time access to the same signed statements as the *Believer*. This fundamental asymmetry in visibility creates opportunities for undetected compromise. This gap is bridged through the introduction of a *Transparency Registry*.

III. Transparency Registries

Transparency registries enable the storage of signed statements in an immutable and append-only manner, ensuring that once a statement has been recorded, it cannot be modified or removed without detection. These registries provide a publicly verifiable history of critical events, which can be retrieved and verified by any interested party.

Within the context of our architecture and the threat model we aim to mitigate, we use transparency registries to record statements produced by *Authors*. These statements include claims about the validity and authenticity of firmware updates. A typical statement is a *signed manifest file* that contains metadata such as the firmware version, the target device model(s), a cryptographic hash of the firmware binary, and optionally a pointer to known vulnerabilities that the update addresses. An example structure for such manifest files is provided in RFC 9124 [2].

The trustworthy operation of transparency registries is ensured through *auditing*: a process during which independent entities periodically check the registry for consistency, correct log behavior, and the inclusion of valid statements. Additionally, *monitoring services* observe the registry for specific events, such as the appearance of statements signed by a particular Author, and notify Authors whenever new entries bearing their signatures appear in the log, helping the detection of unauthorized or suspicious publishing events. Auditing and monitoring can be provided by any entity of the architecture, as well as by third party providers.

A. An improved firmware update architecture

The firmware update process is enhanced to incorporate transparency registries as follows (see also Fig. 2):

1) The *Author* generates a manifest file that includes claims about the firmware (e.g., identity, integrity, applicable device models, and relevant metadata). The manifest is cryptographically signed using the Author's private key. The signed manifest is submitted to the *transparency registry*.
2) The transparency registry appends the statement to its log and returns a *proof-of-inclusion* (PoI). This proof is a new type of signed statement, issued by the registry, attesting that the original statement has been immutably recorded.
3) The firmware binary, the signed manifest, and the PoI are included in a *bundle*, which is uploaded to the *firmware server* for distribution.

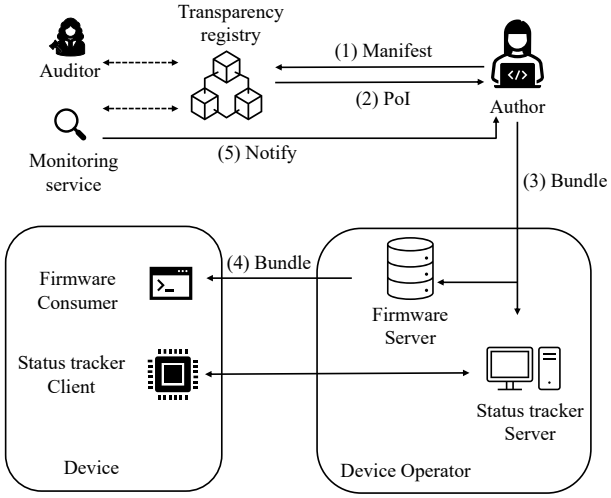

Fig. 2. Our firmware update architecture.

4) When the *firmware consumer* component on an IoT device downloads a firmware update, it retrieves all three artifacts included in the bundle. Also, it verifies the Author's signature on the manifest and also verifies the registry's signature on the PoI, thereby ensuring both the authenticity of the firmware and its proper recording in the transparency log.
5) In parallel, *monitoring services* observe the registry and alert Authors if a manifest signed under their identity appears unexpectedly, enabling rapid detection of key compromise or unauthorized use.

This architecture ensures that firmware updates are not only cryptographically signed but also publicly accountable through verifiable logging, strengthening the resilience of the firmware supply chain. Transparency registries can directly mitigate the attacks described in our threat model by enforcing transparency and traceability of all firmware signing events. Specifically, in order for a firmware—malicious or not—to be considered valid by the system, a corresponding statement must be recorded in the registry. This requirement creates a public and immutable record of every firmware release. As long as *auditors* ensure that the registry functions correctly (i.e., as an append-only, immutable log), and *monitoring services* behave as expected, Authors will be notified whenever a statement is recorded under their identity. If such a notification corresponds to an unauthorized or malicious firmware release, the Author can take immediate corrective action—such as revoking the firmware, publishing an alert, or rotating their signing keys. From the perspective of firmware consumers, trust in a firmware update now depends not only on the Author's digital signature, but also on the successful verification of the associated registry statement and its proof-of-inclusion. As long as the registry is correctly monitored and its integrity is maintained through regular auditing, consumers can rely on it as a trustworthy source of firmware authenticity. In this way, transparency registries close the visibility gap between Verifiers and Believers that was exploited in the attacks

discussed earlier. By ensuring that all parties have access to the same verifiable statements, and by holding registry behavior accountable, the system prevents covert misuse of signing keys and unauthorized firmware distribution.

In this paper, we investigate two types of transparency registries: *centralized registries*, which are maintained by a single trusted authority or organization, and *decentralized registries*, which leverage blockchain technology to distribute trust among multiple parties and eliminate the need for a central operator. Both models are evaluated in the context of securing firmware update process.

B. Centralized Transparency Registries

Centralized transparency registries can be implemented efficiently using cryptographic data structures based on *Merkle trees* [3]. A Merkle tree organizes a collection of signed statements (e.g., firmware manifests) into a binary tree structure where each leaf node corresponds to a cryptographic hash of a statement, and each internal node is the hash of the concatenation of its child nodes. The root of the tree—called the *Merkle root*—serves as a compact commitment to the entire set of statements. Given a Merkle root, a verifier can later validate the inclusion of a specific statement using a *proof-of-inclusion*, (PoI) which consists of the hashes along the path from the leaf node to the root.

A well-established example of such a system is the *Certificate Transparency (CT)* registry, initially defined in RFC 6962 [4] and later extended in RFC 9162 [5]. CT is a scalable, production-grade transparency solution deployed by major Certificate Authorities (CAs) to log the issuance of TLS certificates in an append-only and verifiable log structure. Its design, based on Merkle trees, allows for efficient inclusion of proofs and consistent auditing across large-scale deployments.

The architecture proposed in RFC 6962 has influenced the design of several systems with goals similar to ours, including binary transparency, software update validation, and secure firmware distribution. It provides a foundation for append-only logging, auditability, and decentralized trust in a centralized infrastructure.

However, due to the inherently centralized nature of these registries, the role of external *auditors* becomes crucial for maintaining security and accountability. A single misbehaving registry could equivocate—providing different views of the log to different clients—without detection if not actively monitored. To mitigate this risk, multiple independent *auditors* should be deployed, operating from diverse network vantage points. These auditors continuously observe the log, fetch inclusion and consistency proofs, and compare their views with each other. To detect inconsistencies or misbehavior, they use a *gossip protocol*, which allows them to exchange log-related responses and synchronize their perspective of the registry's state (e.g., [6]). Gossiping prevents tampered or forked versions of the log.

This collective verification model provides strong accountability for centralized transparency registries, even in adversarial settings, and ensures that firmware update statements remain universally visible and verifiable.

C. Blockchain-based registries

Blockchain systems provide a decentralized alternative to centralized transparency registries. In these systems, the registry functionality is implemented through a distributed application, such as a *smart contract* in Ethereum or a *chaincode* in Hyperledger Fabric. These applications manage the storage of transparency statements in a tamper-evident and append-only manner across a network of participating nodes.

A key feature of blockchain-based registries is that all participating nodes inherently act as *auditors*. They collectively ensure the correct execution of the smart contract or chaincode through a *consensus protocol*, which provides strong guarantees about the consistency and integrity of the recorded data. As a result, transparency and correctness are enforced not by a single trusted entity but by a distributed set of nodes that validate every operation. Monitoring in blockchain-based systems is also inherently more robust. Since each new block of transactions is broadcast to all nodes in the network, it is straightforward to observe and validate the entire history of recorded statements from multiple, independent vantage points. This transparency simplifies the implementation of monitoring services, as any party can independently audit the registry by simply following the blockchain.

However, blockchain-based transparency registries also come with trade-offs. Public blockchains, such as Ethereum, may impose *monetary costs* for each transaction (e.g., gas fees), making them less suitable for high-frequency or resource-constrained applications. To avoid such costs, our design adopts a *permissioned blockchain* model, where a network of trusted nodes operates without economic incentives. While this avoids transaction fees, it introduces operational complexity: the blockchain network must be maintained with high availability and fault tolerance to ensure consistent service. Another challenge lies in inclusion verification. Unlike Merkle-tree-based centralized registries where a PoI can be computed and verified offline, blockchain-based registries typically require online interaction. A PoI in this case typically includes information about the corresponding blockchain transaction: a client must query the blockchain (or a node) to confirm the validity of this transaction. This dependency limits the ability of end devices to perform fully offline verification. Furthermore, running a full blockchain node can impose significant computational and communication overhead. In practice, most clients interact with the blockchain through a *gateway node* or *API proxy* that exposes query interfaces. While this reduces the burden on resource-constrained devices, it introduces a potential *single point of failure* or trust bottleneck–if the gateway is compromised or becomes unavailable, the client's ability to verify registry state may be affected.

IV. Implementation

We implement two systems to evaluate our architecture: one based on a centralized transparency registry and another leveraging blockchain technology.

For the centralized variant, we build upon the Sigstore [7] ecosystem and its transparency log service, Rekor[2], which is operated as a public good by the Open Source Security Foundation (OpenSSF). Rekor provides a RESTful API for submitting signed software artifacts, including digital signatures and associated metadata. Specifically, the API accepts an entry that includes: (i) a digital certificate containing the public key (of the firmware Author), (ii) the signature generated over the artifact (e.g., firmware), and (iii) and the hash of the signed data.

Upon submission, Rekor verifies the validity of the provided certificate and responds with a *proof of inclusion*—a signed JSON object, approximately 3,880 bytes in size, which attests that the entry has been (or will be) included in the transparency log. Firmware consumers are expected to verify the signature on the firmware using the public key contained in the certificate, and validate that the proof of inclusion corresponds to the correct certificate, signature, and hash of the firmware.

Rekor's architecture is similar in design to Certificate Transparency systems. It does not immediately incorporate submitted entries into the Merkle tree root; instead, it aggregates multiple submissions and batches them into periodic updates. As a result, the proof of inclusion initially acts as a *promise* that the statement will be included in the log at a future time. In practice, inclusion may be delayed by several minutes. However, current implementations and clients often do not verify whether this promise is eventually fulfilled. This creates a potential security risk, as a misbehaving registry could issue a valid-looking proof without ever including the corresponding entry in the log. Verifiers must therefore be equipped to perform follow-up checks if strong integrity guarantees are desired.

To evaluate registry behavior in practice, we implemented a custom monitor using the `rekor-monitor`[3] tool. This utility allows querying Rekor for entries matching specific conditions, such as certificates with particular `subject` and `issuer` fields. In our proof of concept, we used a custom certificate authority and issued certificates with the `subject` field set to a URL identifying the firmware Author. Using this monitoring setup, we observed that log entries typically appear in the Rekor transparency log approximately 12 minutes after the initial API submission. This validates the expected batching behavior.

For our blockchain-based implementation, we leveraged the *European Blockchain Services Infrastructure (EBSI)*[4], a permissioned blockchain operated by a consortium of trusted nodes across Europe. EBSI is designed to support public-sector applications, offering a secure and scalable infrastructure for cross-border digital services. Our prototype uses EBSI's pilot environment and integrates with its *Track and Trace (TnT) API*[5], which is specifically intended for registering and verifying digital records. To interact with the API, we utilized the `ebsi-services-wrapper`, a TypeScript library provided by the TRACE4EU project[6].

EBSI identifies participants using *Decentralized Identifiers (DIDs)*. A DID is a URL-like identifier that can be resolved to metadata, including the associated public key. In our implementation, firmware *Authors* are each assigned a DID and sign blockchain transactions using the corresponding private key, thereby enabling cryptographic proof of authorship for all actions recorded on the ledger.The EBSI TnT API allows submitting a hash (digest) of a digital artifact along with optional metadata. In our system, Authors register the hash of each firmware update and attach, as metadata, the same certificate and digital signature used in the centralized Rekor-based implementation. This ensures consistency between the two registry models and allows equivalent verification by consumers.

Unlike centralized systems, the EBSI blockchain offers fast inclusion times; we observed that new entries were committed to the ledger within approximately 4 seconds. Firmware consumers can later query the registry to confirm the presence of a specific firmware hash, thereby validating the integrity and authenticity of the update. To complement this functionality, we implemented a custom monitoring service for EBSI. The TnT API supports listing recent registrations, enabling continuous observation of the blockchain. Our monitor checks each new record and determines whether it was submitted by the firmware Author's DID. This mirrors the behavior of our Rekor-based monitor and ensures that both implementations offer comparable transparency and detection capabilities.

V. Related work

Several related efforts employ transparency services to secure the software supply chain. The work in [8] leverages the Bitcoin blockchain to provide verifiable software traceability, pursuing goals closely aligned with our own architecture. In contrast, our blockchain-based variant utilizes a permissioned blockchain, which avoids monetary costs and offers significantly better performance. Similarly, the approach in [9] uses transparency registries to record and enforce software supply chain policies. Our work is complementary to this, as firmware validation policies can also be captured and logged in a transparency registry following a similar methodology. In another line of work, [10] proposes the use of transparency registries to log source artifacts related to code executed within confidential computing environments. While this approach shares both goals and foundational techniques with ours—such as verifiable logging and statement auditing—it is applied in a different context. Together, these works highlight the growing relevance and versatility of transparency mechanisms in securing various facets of modern software ecosystems.

[2]https://rekor.sigstore.dev/

[3]https://github.com/sigstore/rekor-monitor/

[4]https://ec.europa.eu/digital-building-blocks/sites/display/EBSI/Home

[5]https://hub.ebsi.eu/apis/pilot/track-and-trace/v1

[6]https://github.com/trace4eu/ebsi-services-wrapper

Solutions based on transparency services and traceability are actively being researched, particularly in the domain of *key transparency* (e.g., [11]–[13]). These systems have gained significant traction and are already deployed at large scale in real-world applications. For instance, CONIKS [14], a key transparency system designed for end-to-end encrypted communication, has been adopted to provide transparency for Apple's iMessage service. Transparency registries are also being adopted to secure a variety of emerging system architectures. Duan et al. [15] propose a secure Internet naming system that incorporates transparency services to enhance trust and resilience. Schaefer et al. [16] utilize a blockchain-based transparency log to audit access to personal data during AI model training, providing accountability in privacy-sensitive contexts. In the domain of IoT, Wan et al. [17] design a Certificate Transparency system specifically tailored to the unique constraints of IoT environments. Meanwhile, Mei et al. [18] apply transparency mechanisms to protect web applications from misconfigurations and supply chain attacks. These works demonstrate the broad applicability of transparency technologies across domains.

While transparency registries enhance trust and accountability, their public nature can also introduce new security risks. Recent work [19] has shown how malicious bots leverage Certificate Transparency logs to rapidly discover newly issued certificates and identify potential targets for attacks. In response, emerging research explores new cryptographic data structures that aim to balance traceability with enhanced user privacy (e.g., [20]). Similarly, recent work has shown that monitoring the Certificate Transparency services introduces significant overhead and improvements are proposed [21]. These approaches offer promising directions for improving transparency systems. Our solution stands to benefit from these advances, potentially mitigating the scalability, security and privacy concerns associated with fully public registries.

VI. Conclusion

In this paper, we addressed the security risks associated with firmware signing, particularly those arising from compromised signing keys. We proposed an architecture that integrates transparency and traceability through the use of transparency registries, i.e., tamper-proof logs where all signing events are recorded and monitored. We implemented two variants of our system: one based on a centralized transparency registry using Sigstore's Rekor, and another leveraging the permissioned blockchain infrastructure provided by EBSI. Our results demonstrate that both approaches are practical and effective for enhancing trust in firmware updates, with trade-offs in terms of latency, cost, and operational complexity. Future work will focus on improving the robustness of monitor infrastructure and extending our system to accomodate other types of software artifacts.

References

[1] B. Moran, H. Tschofenig, D. Brown, and M. Meriac, "A Firmware Update Architecture for Internet of Things," RFC 9019, Apr. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9019

[2] B. Moran, H. Tschofenig, and H. Birkholz, "A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices," RFC 9124, Jan. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9124

[3] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa, "Merkle 2: A low-latency transparency log system," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 285–303.

[4] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962, Jun. 2013. [Online]. Available: https://www.rfc-editor.org/info/rfc6962

[5] B. Laurie, E. Messeri, and R. Stradling, "Certificate Transparency Version 2.0," RFC 9162, Dec. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9162

[6] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities" honest or bust" with decentralized witness cosigning," in *2016 IEEE Symposium on Security and Privacy (SP)*. Ieee, 2016, pp. 526–545.

[7] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software signing for everybody," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2353–2367.

[8] M. Al-Bassam and S. Meiklejohn, "Contour: A practical system for binary transparency," in *International Workshop on Data Privacy Management*. Springer, 2018, pp. 94–110.

[9] A. Ferraiuolo, R. Behjati, T. Santoro, and B. Laurie, "Policy transparency: Authorization logic meets general transparency to prove software supply chain integrity," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 2022, pp. 3–13.

[10] A. Delignat-Lavaud, C. Fournet, K. Vaswani, S. Clebsch, M. Riechert, M. Costa, and M. Russinovich, "Why should I trust your code?" *Communications of the ACM*, vol. 67, no. 1, pp. 68–76, 2023.

[11] H. Malvai, L. Kokoris-Kogias, A. Sonnino, E. Ghosh, E. Oztürk, K. Lewi, and S. Lawlor, "Parakeet: Practical key transparency for end-to-end encrypted messaging," *Cryptology ePrint Archive*, 2023.

[12] J. Len, M. Chase, E. Ghosh, K. Laine, and R. C. Moreno, "{OPTIKS}: An optimized key transparency system," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4355–4372.

[13] J. Len, M. Chase, E. Ghosh, D. Jost, B. Kesavan, and A. Marcedone, "ELEKTRA: Efficient Lightweight multi-dEvice Key TRAnsparency," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2915–2929.

[14] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "{CONIKS}: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 383–398.

[15] H. Duan, R. Fischer, J. Lou, S. Liu, D. Basin, and A. Perrig, "{RHINE}: Robust and high-performance internet naming with {E2E} authenticity," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 531–553.

[16] C. Schaefer and C. Edman, "Transparent logging with hyperledger fabric," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 65–69.

[17] H. Wan, Q. Wang, Y. Teng, C. Ma, J. Lin, and M. Wang, "Imct: A feasible scheme for deploying implicit certificates with certificate transparency in iot," in *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, 2023, pp. 1–10.

[18] E. Meißner, F. Kargl, and B. Erb, "Wait: protecting the integrity of web applications with binary-equivalent transparency," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ser. SAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1950–1953.

[19] B. Kondracki, J. So, and N. Nikiforakis, "Uninvited guests: Analyzing the identity and behavior of certificate transparency bots," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 53–70.

[20] D. Reijsbergen, A. Maw, Z. Yang, T. T. A. Dinh, and J. Zhou, "{TAP}: Transparent and {Privacy-Preserving} data services," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6489–6506.

[21] A. Sun, J. Lin, W. Wang, Z. Liu, B. Li, S. Wen, and Q. W. F. Li, "Certificate transparency revisited: The public inspections on third-party monitors," in *NDSS*, 2024.