International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V73I11P106

Volume 73 Issue 11, 37-42, November 2025
© 2025 Seventh Sense Research Group®

Original Article

Leveraging Generative AI for Intelligent Code Signing and Tamper Detection

Karthikeyan Thirumalaisamy

Independent Researcher, Washington, USA.

Corresponding Author : kathiru11@gmail.com

Received: 20 September 2025 Revised: 28 October 2025 Accepted: 15 November 2025 Published: 29 November 2025

Abstract *- Code signing is a fundamental tool for establishing trust and integrity in today's software ecosystems. However, as traditional code-signing methods are largely static, these methods only provide a guarantee based on the certificate issuer and validity at the time of signing, without regard to the contextual behavior associated with the signing process. As such, they are vulnerable to various factors, such as insider abuse, unauthorized access to signing keys, and post-signing tampering. This paper presents a novel paradigm - Leveraging Generative AI for Intelligent Code Signing and Tamper Detection - which is built on an AI-based Trust Graph framework. The proposed work utilizes Generative Artificial Intelligence (Generative AI) and Graph Neural Networks (GNNs) to model the dynamic relationships between the developers, the repositories, the certificates, and the binaries, allowing for the detection of abnormal signing behavior patterns that may indicate compromise. The system continuously analyzes behavioral patterns as well as provenance-type data to achieve AI-based trust scoring and contextual anomaly detection to uncover instances of unauthorized use of keys, insider tampering, and subtle compiler-related or code modification instances, which are typically missed by static methods. The adaptive trust ecosystem enhances both the integrity of the software and resilience in the software supply chain, and clearly demonstrates the way in which generative AI can bridge the gap between traditional authenticity verification and real-time threat mitigation.*

Keywords *- Code Signing, Generative AI, Code Integrity, Supply Chain Security, Insider Threats, AI Security, GNN.*

1. Introduction

The security and trustworthiness of the software supply chain have become a central area of concern, given the fact that software is the cornerstone for basic industries such as finance, health care, energy, and defense. Digital signing has, for many years, provided the fundamental model for asserting authenticity and ensuring that the wide distribution of software is from a legitimate source. However, with the development of software environments to become complex interrelated systems of developers, build systems, repositories, and third-party components, the code signing model has shown its own weaknesses. These various trust relationships can now be exploited both from without and within, by competitors, to inject evil code, compromise signing keys, or otherwise to repurpose genuine product without detection. A growing number of high-profile supply-chain hacking attacks, such as SolarWinds and XZ Utils, have shown the grievous nature of compromised signing workflows and unauthorized exploitation of trusted certificates.

Established code signing processes have relied upon static validation mechanisms to confirm that a digital signature is associated with a trusted certification authority. This cryptographic verification provides confirmation of authenticity at a particular point in time, but gives the user no context regarding the origin of how a binary was arrived at, the identity of who signed it, and whether their activity conformed in any way to known accepted norms. For this reason, present-day code signing technologies are not in a position to adequately deal with insider exploitation, key compromises, or alteration of products post-signing, all of which spoil the whole purpose of the trust that these tools are supposed to establish.

This paper proposes a new paradigm: Leveraging Generative AI for Intelligent Code Signing and Tamper Detection, which revolutionizes the process of how software authenticity and trust in terms of behavior patterns are certified across the software supply chain. The mechanism thus proposed combines generative artificial intelligence with graph neural networks to create a Trust Graph, a dynamic algorithmic model, which contains the many-to-many relationships between developers, repositories, certificates, and binaries. By continually monitoring previous signing activity history, capturing contextual metadata associated therewith, it is possible for the Trust Graph to identify anomalous events which may indicate misuse of a key, insider manipulation, or injection of malevolent components.

The essential point of this paper is that code signing has to evolve from a static verification method into an adaptable intelligent trust framework. Due to AI-driven trust scoring, contextual detection of anomalies, and continual provenance of validation, a proactive defense against insider threat and supply-chain exploitation is to be established through the mechanism covered in this paper. It will also reveal how generative AI will serve the purpose of predictive enforcer of trust, being capable of isolating early indicators of compromise and thus maintaining trust in the authenticity of software artifacts passed through build and deployment pipelines. This will state a self-learning trust environment, capable of not only verifying what is signed in terms of software but also how and by whom it is signed in terms of verification. This will bridge the dichotomy between assurance of authenticity and real-time threat monitoring, inherent in modern software environments.

2. Traditional Code Signing Approach

Code signing is a fundamental security measure designed to authenticate and protect the integrity of code before it is executed. It utilizes Public Key Infrastructure (PKI), in which developers sign code with a private key of their own creation and are issued a public certificate to distribute as proof of authenticity. This means the software originates from a trusted source and undergoes the same verification process, ensuring that it has not been modified in any way since it was signed.

The traditional form of code signing is widely used in various parts of the software world, including Microsoft's Authenticode, Apple's Gatekeeper, and numerous Linux package managers, to facilitate trusted software distribution channels. This means that while the whole process is effective for checking whether the software is valid or not, it does not have the ability to be effectively contextual. Once a signature has been verified as valid, it will continue to be trusted indefinitely, even if the signing key is later compromised or misused. As a result, attackers, especially insiders or those with stolen credentials, can exploit this static trust model to distribute malicious yet cryptographically valid software. Thus, the static application of trust can be abused through the distribution of software that is properly signed with cryptography, which is otherwise malicious.

2.1. How Code Signing Works?

Software authenticity and integrity have depended on code signing as their fundamental security measure for many years. The main purpose of code signing is to verify that software artifacts come from authorized sources and their content remains unchanged since their initial release.

The validation process follows a basic sequence of cryptographic operations.

- The developer uses their private key to encrypt the hash function output, which results in the digital signature.
- A software artifact undergoes hash generation to produce a digital fingerprint.
- The digital signature becomes verifiable through public key validation, which depends on certificates issued by trustworthy Certificate Authorities (CAs). The software publisher uses Public Key Infrastructure (PKI) to generate digital signatures for executables, scripts, and packages through private key encryption.
- The artifact requires both a public key and a CA-issued certificate to be attached to it.
- The recipient system performs public key decryption to verify the signature before it generates a new hash for verification against the original hash.
- The software proves its authenticity through hash verification, which shows no evidence of modifications.

2.2. Limitations of Traditional Code Signing

The traditional code signing process provides immediate cryptographic authentication through its signing mechanism, but it fails to meet the requirements of modern distributed enterprise software systems.

The current software development methods, which include continuous integration and cloud-based builds, open-source dependencies, and software signing pipelines, have demonstrated that the static trust model is ineffective.

2.2.1. Static, One-time Validation

Code signing makes authentication checks only once, at the time of installation or execution, and it is presumed trustworthy for all time unless the certificate is revoked. There is no ongoing or behavioral validation intended to ensure the software remains trustworthy.

2.2.2. Provider Exposure

If the development provider's private key is stolen or used by an untrustworthy party, the hacker can sign malware as permanent or verified binaries that show up on any verification system as being absolutely legitimate and verified by their source. Such incidents are frequently not detected until a manual supervisory control discovers them.

2.2.3. Absence of Contextual Awareness

The signing process makes certain only that there is cryptographic correctness, but does not detect who signed it, under what conditions, or whether this was consistent with normal organizational behavior. There is always the potential for the misuse of internal parties or for the misuse of signing credentials by automated means. In any case, internal misuse of signatures is invisible to any existing supervision of the mechanism.

2.2.4. Tampering Post-Signing

The signing process in unsecured build environments allows hackers to modify or insert code after signing occurs, before the code reaches its packaging stage. The current verification methods fail to detect any modifications made to the code after the signing process has occurred.

2.2.5. Failure to Detect Internal Threats

Since all valid signatures are treated alike, internal users with legitimate access can misuse their access intentionally without any alert raised. The validity of valid certificates creates an environment of blind trust, making it difficult to distinguish between legitimate and illegitimate activity. These problems expose a fundamental weakness in code signing, which is that it deters against authenticity rather than trustworthiness. In enterprise applications that are increasingly adopting more complex DevOps and CI/CD practices, static verification is no longer sufficient. Instead, it is necessary to have a model that is more intelligently dynamic and is designed to continually ascertain relationships, behavior, and patterns, leading to the analysis of signing.

3. AI-Driven Trust Graph for Intelligent Code Signing

This study addresses the static and context-blind issues with existing code signing through an AI-Driven Trust Graph Framework, which utilizes Generative Artificial Intelligence (AI) and Graph Neural Networks (GNNs) to provide dynamic and behaviorally aware authentication for software integrity.

In this research, the authors have transformed the current code signing process into a learning based, real-time trust model that can detect anomalies in the signing activity (including insider misuse), post-signing tampering, etc., as they occur.

As such, at the heart of the framework are the Trust Graphs - a data structure that captures the complexities of developer relationships, certificate relationships, repository relationships, binary relationships, and build system relationships. Each node in the graph represents an entity within the signing environment, and each edge captures the historical and behavioral connections between nodes. Through continuous learning about the historical connections, the system builds a baseline of normal signing behavior; thus, when a deviation occurs, the system will recognize this deviation as a possible indication of malicious or inappropriate use.

3.1. Architecture

The proposed AI-Driven Trust Graph Framework architecture combines generative artificial intelligence with Graph Neural Networks (GNNs) and integrity scoring mechanisms to perform intelligent code-signing activity validation. The system operates as a multi-level framework that monitors build and signing operations and provenance data to identify irregularities while assessing trust levels in real-time. The system design is depicted in Figure 1.

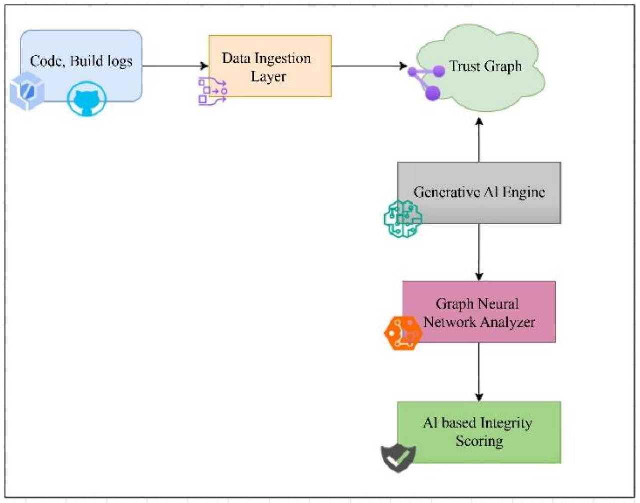

Fig. 1 Illustration of AI-driven Trust Graph architecture

3.1.1. Code and Build Logs

It begins with the build systems, which are the automation layers (that compile, test, and prepare for release) for building software. Most modern development environments utilize Continuous Integration / Continuous Delivery (CI/CD) Pipelines like Jenkins, GitHub Actions, Azure DevOps, GitLab CI, etc., to run a sequence of automated steps for each code change – from retrieving the Source Code to resolving dependencies, compiling, testing, creating artifacts, and preparing signing requests.

As the build systems execute these processes, they will collect an abundance of telemetry and metadata that provides the complete context of software creation. This can include, but is not limited to, the commit ID, developer identity, repository/branch, timestamps for builds, dependency lists/SBOMs, build environment parameters, and any signing requests generated when creating the artifacts. All of this information forms a verifiable record of who built what, when, and how.

After the build process has completed, this metadata is automatically sent to the Data Ingestion Layer via secured connectors or event streams. The data collected will be used as the foundation for subsequent analysis, allowing the AI components of the system to reason about the provenance of code, verify the integrity of builds, and later correlate the behavior of developers and artifacts in the Trust Graph.

3.1.2. Data Ingestion Layer

Data is ingested at the Data Ingestion Layer, which acts as an aggregation point (i.e., central repository) to collect, validate, and organize metadata from all parts of the software supply chain. Data is aggregated from build pipelines, version control systems, artifact repositories, signing servers, and certificate authorities, and collects and stores the following critical pieces of data: identity of the developer(s), reference to the commits, how/when the signing keys were used, and origin (provenance) of each artifact.

Data once collected is normalized and validated to ensure consistency and accuracy of the source data. Normalization encompasses the alignment of schemes, synchronization of timestamping, resolution of identities, and validation of the integrity of source metadata using cryptographic techniques and methods to prove the origin of the source data (provenance). Validation ensures the quality of the data being ingested into the system (i.e., it is complete, trustworthy, and has not been tampered with). The normalized and validated dataset is then passed to the Graph Construction Model, which creates a Trust Graph, i.e., a graph that represents the relationships between entities within the software ecosystem based on the verified data. The Trust Graph includes entities represented by nodes (e.g., developers, repositories, certificates, and binaries), and relationships between these entities are represented by edges (e.g., "developer signed binary" or "CA issued certificate").

The Trust Graph provides an entity-based view of the relationships between all entities involved in the software build and signing processes. The Trust Graph provides the foundation upon which AI components can perform behavior analysis, identify anomalies, and serve as the basis for determining intelligent integrity scores in subsequent layers of the framework.

3.1.3. Generative AI Engine

The Generative AI Engine serves as the analytical core of the system, creating models that extract standard signing behaviors from the Trust Graph. The Generative AI Engine examines how developer-repository-certificate-binary entities interact with each other to identify standard organizational software component signing methods.

The AI Engine reviews past signing activities of entities involved in the signing process to create behavioral models for both specific signers and their respective projects. The Generative AI Engine creates models to monitor typical signing behavior of signers and projects through time by processing context data from signing events. The system uses acquired baseline information to detect both standard signing activities and potential security incidents, insider threats, and compromised credentials.

The AI Engine detects signing events that originate from unidentified build agents during unverified time periods with unidentifiable dependencies as suspicious activities that require additional evaluation.

3.1.4. Graph Neural Network (GNN) Analyzer

The Graph Neural Network (GNN) Analyzer is also a core element of the overall framework, which analyzes data from the Trust Graph (which has been enhanced by the Generative AI Engine). The generative model develops behavioral baselines of what is expected, while the GNN discovers relational anomalies (i.e., irregularities in how entities relate to one another) within the software signing environment. When processing the nodes and edges of the graph, the GNN captures the contextual relationships of all entities, including developers, signing keys, repositories, certificates, and artifacts. It learns the structural and temporal dependencies that define legitimate trust paths, allowing it to identify small deviations from these established trust norms. In doing so, it detects such events as when a developer signs using a previously unknown signing key for a different repository; when signing occurs outside of the developer's usual operating hours; or when there is an unusual spike in signing activity, suggesting possible misuse of automation or leakage of credentials. Likewise, it identifies instances where a certificate is used across multiple unrelated projects, indicating potential policy violations or compromised keys.

Each event is evaluated based on its neighbors and previous interactions. Through the propagation of information via the graph structure, the GNN measures how much a specific relationship deviates from its baseline behavior, effectively measuring the degree of anomaly for that relationship. These relational findings offer a multidimensional understanding of the changes that have occurred in the signing environment and how they evolved. Through ongoing relational analysis of the GNN Analyzer, the system is better able to detect complex, cross-entity threats typically not visible to traditional verification methods. The relational findings from the GNN analyzer are input into the AI-based integrity scoring module, where the deviation is weighted to assess the trustworthiness of each signing activity dynamically.

3.1.5. AI-Based Integrity Scoring

The AI-Based Integrity Scoring Module functions as the central decision-making component within the framework, integrating behavioral and cryptographic insights to assess trustworthiness. The module's function is to integrate insights from the Generative AI Engine and the Graph Neural Network (GNN) Analyzer into a single, quantifiable measure of software integrity, referred to as the Trust Score. Integrity is assessed for each signing event, build action, or artifact release using a multi-dimensional approach. Each event is analyzed within the three primary dimensions of software integrity:

Behavioral Consistency

Evaluates the degree to which the signing activity deviates from historical patterns. This includes examining such factors as the developer's typical signing time, build environment, and dependency context. Deviation in these areas may be indicative of insider misuse or compromised automation.

Cryptographic Validity

Verifies that the artifact's digital signature, certificate chain, and timestamp integrity have not been compromised. Ensures that an authorized signer owns the private key used, that the certificate has not expired nor been revoked, and that the signing order complies with established policies.

Source Integrity

Utilizing data and metadata from the Data Ingestion Layer and Trust Graph, this dimension ensures that the artifact's lineage - its source repository, build environment, and dependency chain that corresponds to confirmed provenance records. Identifies instances in which the build or signing occurred outside of designated environments or when dependencies diverged from established baselines.

The Integrity Scoring Module then combines the results of the three dimensions to create a comprehensive Trust Score that reflects the overall confidence in the legitimacy and integrity of the software artifact. High levels of trust indicate normal, policy-compliant signing activity; low levels of trust identify possible tampering, unusual signing behavior, or unauthorized use of credentials. Through the application of an adaptive scoring model to quantify integrity, the Integrity Scoring Module transforms code-signing validation from a static, one-time validation to a continuous and contextualized process for assessing trust in the software artifact.

The resulting Trust Scores are then passed back to the feedback loop, enabling the system to learn from identified security events, update baselines, and continually improve predictive capabilities for newly emerging threats.

3.1.6. Feedback Loop: Integrity Scoring

The Feedback Loop serves as the adaptive intelligence system of the AI-Driven Trust Graph framework, utilizing real-world validation results to facilitate continuous improvements. The AI-Based Integrity Scoring Module generates trust scores, anomaly reports, and alerts, which get sent back to the Data Ingestion Layer to enhance the system's knowledge about behavioral patterns and relationships.

The system uses a continuous data flow, which turns all detection results into new learning data for the framework. The framework updates its Generative AI Engine and GNN Analyzer through model retraining when it verifies events with high trust scores and when it confirms incidents or detects anomalies. The system develops the ability to distinguish between authorized unusual activities and actual security threats through its learning process.

The feedback loop enables the framework to evolve into a self-enhancing trust system, operating as a dynamic validation mechanism. The system enhances its ability to detect anomalies and assess trust through continuous adaptation to developer actions, build environment changes, and new attack methods.

4. Conclusion

The research introduces an AI-driven trust graph framework, which enhances software authentication and integrity verification operations in contemporary development systems. The framework utilizes generative AI in conjunction with Graph Neural Networks (GNNs) and adaptive integrity scoring to develop an intelligent trust model that undergoes continuous evolution. The system maintains ongoing software authenticity checks by analyzing code signing originators, development sites, publication dates, and previous signing activities. The system detects vital changes in signing activities, which reveal possible credential exploitation or insider threats, or post-signing alterations that traditional static verification tools cannot detect.

The framework maintains its operation through a feedback system, which prevents stagnation because validation results from normal or suspicious activities help the system improve its ability to recognize behavioral patterns and understand relationships. The system learns to detect genuine security threats by studying normal development patterns throughout time.

The proposed framework develops an intelligent trust management system that learns from experience to boost its accuracy and protection capabilities during software environment transformations. The system generates an adaptive defense mechanism that implements proactive security measures based on context to protect contemporary software supply chain.

References

[1] Tiantian Ji et al., "Scrutinizing Code Signing: A Study of in-Depth Threat Modeling and Defense Mechanism," *IEEE Internet of Things Journal*, vol. 11, no. 24, pp. 40051-40069, 2024, [CrossRef] [Google Scholar] [Publisher Link]

[2] Adrian Brodzik, and Wojciech Mazurczyk, "AI Model Signing for Integrity Verification," *2025 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, 2025. [CrossRef] [Publisher Link]

[3] Platon Kotzias et al., "Certified PUP: Abuse in Authenticode Code Signing," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 465–478, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[4] Doowon Kim, Bum Jun Kwon, and Tudor Dumitras, "Certified Malware: Measuring Breaches of Trust in the Windows Code-signing PKI," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1435–1448, 2017. [CrossRef] [Google Scholar] [Publisher Link]
[5] Digicert, What is Code Signing?. [Online]. Available: https://www.digicert.com/faq/code-signing-trust/what-is-code-signing
[6] Microsoft, Use Code Signing for Added Control and Protection with App Control for Business. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/deployment/use-code-signing-for-better-control-and-protection
[7] AWS, Using Code Signing to Verify Code Integrity with Lambda. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-codesigning.html
[8] Microsoft, Platform Code Integrity. [Online]. Available: https://learn.microsoft.com/en-us/azure/security/fundamentals/code-integrity
[9] Microsoft, Code Integrity Checking. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/code-integrity-checking
[10] Qodo, Code Integrity. [Online]. Available: https://www.qodo.ai/glossary/code-integrity/
[11] IBM, What is a GNN (Graph Neural Network)?. [Online]. Available: https://www.ibm.com/think/topics/graph-neural-network
[12] Nvidia Developer, Graph Neural Network Frameworks. [Online]. Available: https://developer.nvidia.com/gnn-frameworks
[13] Amal Menzli, Graph Neural Network and Some of GNN Applications: Everything You Need to Know, 2025. [Online]. Available: https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications
[14] Yasmine karray, Explainable AI for Graph Neural Networks, 2024. [Online]. Available: https://medium.com/@ykarray29/explainable-ai-for-graph-neural-networks-a4b89c89983a
[15] Dan Shalev, Can Graph Neural Networks Actually Help With LLM Hallucinations?, 2025. [Online]. Available: https://www.falkordb.com/blog/graph-neural-networks-llm-integration/
[16] Shohil Kothari, Graph Neural Networks and Generative AI, 2023. [Online]. Available: https://www.fiddler.ai/blog/graph-neural-networks-and-generative-ai