Signé n'est pas sûr : mesurer la frontière entre intégrité cryptographique et confiance sémantique dans les chaînes d'approvisionnement de l'IA

Karan Deshmukh, Sudheer Reddy Jogi, Dola Krishna Dama, Rambabu Mekala
MSc Cybersécurité et Science des données, ESAIP École d'Ingénieurs, Angers, France
Projet Applicatif 2 (projets d'IA appliquée), encadré par Sofiane Hamrioui
Auteur correspondant : domail2karandeshmukh@gmail.com

***Résumé*—Un modèle déployé est le produit de jeux de données, de poids pré-entraînés et d'affinages venant de parties différentes ; or les contrôles de chaîne d'approvisionnement logicielle signent une compilation ou un fichier sans rien dire de cette filiation ni du sens d'une signature valide pour le comportement du modèle. Nous présentons AI-ChainGuard, qui donne à chaque artefact d'un pipeline d'IA, du jeu de données à l'enregistrement de déploiement, un manifeste signé contenant les empreintes des manifestes de ses parents : un graphe de provenance chaîné par hachage, soumis à dix contrôles, avec confiance et révocation dans un journal signé, ancré et en ajout seul, et une passerelle qui vérifie toute la chaîne au déploiement avant de charger le modèle. Nous *mesurons* ensuite la frontière de ces garanties sur 48 variantes (attaques et témoins) en sept expériences. Toute manipulation de la suite postérieure à la signature, d'un poids modifié de 10^{-6} à la troncature du journal de confiance, est détectée en 1,7 à 15,1 ms chaîne comprise. Mais un jeu de données empoisonné *avant* la signature est déployé en VÉRIFIÉ avec 77 % de succès de porte dérobée, un déclencheur adaptatif contourne les heuristiques de contenu avec 51 %, et le modèle à porte dérobée d'un éditeur de confiance est vérifié avec 100 %. L'attestation coûte 0,8 % du temps d'entraînement, 8,5 ms par vérification, rien par inférence et 1,4 à 3,1 Ko par manifeste, et s'exporte en in-toto/DSSE avec prédicat ChainGuard ou SLSA. Les contrôles de chaîne d'approvisionnement de l'IA doivent rapporter séparément preuves cryptographiques et sémantiques ; code, tests et scénarios sont publiés.**

***Mots-clés*—attestation, chaîne d'approvisionnement de l'IA, chaîne d'approvisionnement logicielle, empoisonnement des données, in-toto, porte dérobée, provenance des modèles, révocation, signatures numériques, SLSA.**

I. Introduction

Les systèmes d'apprentissage automatique sont assemblés plutôt qu'écrits : un modèle en production descend d'un jeu de données collecté par une équipe, d'un modèle pré-entraîné téléchargé depuis un dépôt public, d'un affinage chez un tiers et d'un empaquetage, et chaque passage de main est une occasion de manipulation. En décembre 2022, une confusion de dépendances a remplacé `torchtriton`, dépendance des versions nocturnes de PyTorch, par un paquet exfiltrant clés SSH et fichiers [1]. En février 2024, JFrog a trouvé une centaine de modèles Hugging Face dont la sérialisation *pickle* exécutait du code au chargement [2] ; en 2025, ReversingLabs a décrit des flux pickle corrompus à dessein qui exécutaient leur charge utile avant l'échec du scanner [3]. L'empoisonnement coûte peu : une étude de 2025 a posé une porte dérobée dans des modèles de langue de 600 millions à 13 milliards de paramètres avec un nombre quasi constant de 250 documents empoisonnés [4], et Carlini et al. ont montré qu'empoisonner des jeux de données à l'échelle du web coûterait quelques dizaines de dollars [5]. L'article 15(5) du règlement européen sur l'IA exige des mesures contre « les attaques visant à manipuler le jeu de données d'entraînement (empoisonnement des données) ou les composants préentraînés utilisés en entraînement (empoisonnement de modèle) » [6] ; le NIST AI RMF [7] et MITRE ATLAS (AML.T0010) [8] décrivent le même risque.

La réaction naturelle est de réutiliser la sécurité des chaînes d'approvisionnement logicielles : SLSA [9], in-toto [10] et Sigstore [11] donnent aux compilations une provenance vérifiable, OpenSSF Model Signing étend Sigstore aux modèles [12]. Deux lacunes demeurent. Ces outils attestent une compilation ou un fichier : rien ne relie le jeu de données au modèle de base, celui-ci au modèle affiné et ce dernier au déploiement, avec une politique sur qui signe quoi et une révocation qui se propage le long des liens. Et une signature prouve que des octets sont ce qu'un détenteur de clé a attesté ; pourtant les utilisateurs des dépôts de modèles lisent *signé* comme *sûr*. Pour du code, qui se relit, c'est tolérable ; pour un tenseur de poids, non, et, à notre connaissance, personne n'avait mesuré à quel point c'est faux.

Cet article apporte (1) AI-ChainGuard, une chaîne d'attestation sur quatre étapes avec un vérificateur récursif de dix contrôles, une politique de signataires liée aux rôles, une passerelle de déploiement et un *journal de confiance* signé, ancré et en ajout seul, qui sort la base de données de la base de confiance ; (2) une suite reproductible de 48 variantes (attaques et témoins) en sept expériences, de l'altération après signature et de la réécriture de provenance aux contournements visant le journal lui-même, aux éditeurs non fiables et à l'empoisonnement avant signature ; (3) la mesure de la frontière : toute manipulation postérieure à la signature est détectée, tandis que du poison signé est vérifié avec 77 à 100 % de succès de porte dérobée ; (4) le coût de l'attestation et son interopérabilité (in-toto/DSSE, SLSA).

II. Contexte et travaux connexes

A. Menaces le long d'un pipeline d'IA

Nous distinguons les menaces selon le *moment* où elles agissent. Après la signature, un attaquant ayant accès au stockage, à un miroir ou au réseau peut modifier des octets, substituer un artefact, réécrire l'ascendance pour qu'un modèle semble descendre d'un jeu de données approuvé, forger ou rejouer des signatures, utiliser une clé compromise ou révoquée, ou altérer le registre de confiance ; tout cela est, en principe, à la portée de la cryptographie. Avant la signature, un attaquant qui contrôle la source des données, l'éditeur d'un modèle pré-entraîné ou l'hôte d'entraînement place une porte dérobée [13] qu'un signataire légitime atteste de bonne foi. La sérialisation malveillante [2], [3] est entre les deux : le fichier signé est intact, et il exécute du code au chargement.

B. Signature, provenance et transparence

Sigstore remplace les clés de longue durée par des certificats éphémères liés à des identités OIDC (Fulcio) et consigne chaque signature dans un journal de transparence public (Rekor) [11]. in-toto décrit une chaîne d'approvisionnement comme une disposition (*layout*) d'étapes confiées à des acteurs autorisés (*functionaries*) et vérifie les métadonnées de chaque étape [10] ; son cadre d'attestation définit l'*énoncé* (empreintes des sujets, prédicat typé) et l'enveloppe DSSE utilisée par Sigstore et GitHub [14], [15]. SLSA v1.1 (avril 2025) définit des niveaux de provenance et le prédicat consommé par les moteurs de politique [9] ; OpenSSF Model Signing v1.0 (avril 2025) hache chaque fichier d'un répertoire de modèle dans un manifeste signé par Sigstore ou PKI [12]. The Update Framework ancre un rôle racine et protège du retour arrière [16] ; Certificate Transparency rend toute altération d'un journal détectable grâce à un arbre de Merkle [17]. Cartes de modèle [18] et fiches de données [19] décrivent les artefacts sans les signer ; safetensors retire l'exécution de code du chargement [20]. Les défenses de contenu (Neural Cleanse [21], STRIP [22], signatures spectrales [23], regroupement d'activations [24]) détectent certaines portes dérobées et se contournent. Le tableau I situe AI-ChainGuard : il ne remplace pas ces outils, il relie les artefacts qu'ils signent, impose qui signe quoi et dit ce qu'une signature établit ou non.

III. Conception du système

A. Modèle de menace

De confiance : les clés de signature de l'organisation (dont la clé racine), le petit fichier d'ancrage du journal, la politique et le code du vérificateur. Hostiles : stockage, index, jeux de données, dépôts de modèles externes, hôtes d'entraînement et réseau. L'attaquant peut lire et écrire tout artefact, manifeste, ligne d'index ou enregistrement du journal, rejouer ou forger des enveloppes, détenir une clé révoquée ou publier sous une identité inconnue. Hors périmètre : compromission des composants de confiance, canaux auxiliaires, justesse du code d'entraînement. Le vérificateur recalcule donc chaque hachage, empreinte et racine de Merkle au lieu de croire les valeurs stockées, et ne lit dans la base rien qu'il ne puisse confirmer autrement.

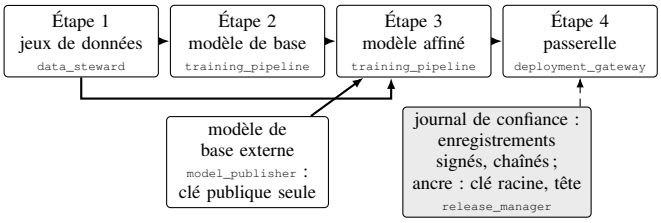


FIG. 1. Quatre étapes attestées et leur rôle signataire. La charge utile signée de chaque enfant contient les empreintes des manifestes de ses parents (flèches pleines). Un modèle de base interne exige un parent jeu de données ; un modèle externe, sans filiation interne, n'est signé que par une clé d'éditeur enregistrée. La passerelle vérifie de nouveau toute la chaîne ; sa vérification, comme toute autre, rejoue le journal de confiance (pointillés).

B. Manifestes et chaîne par hachage

Chaque artefact reçoit un corps de manifeste m : type et étape, SHA-256 et taille du fichier, pour les jeux de données CSV la racine de Merkle des hachages de lignes, la liste des parents, une provenance libre (configuration, environnement), le résultat des contrôles de contenu antérieurs à l'attestation, et l'identité, le rôle et l'identifiant de clé du signataire. Le corps est sérialisé en JSON canonique (clés triées, sans espaces superflus) et signé en entier avec Ed25519 [25] :

$$\sigma = \mathrm{Sign}_k\big(\mathrm{canon}(m)\big), \qquad d(m) = \mathrm{SHA256}\big(\mathrm{canon}(m)\big). \tag{1}$$

Signer tout le corps plutôt que le seul hachage du fichier empêche de garder un couple (hachage, signature) valide tout en changeant l'étape, les parents ou le rôle. L'empreinte $d(m)$ est l'identité de l'artefact dans le graphe : un enfant inscrit chaque parent sous la forme $(\mathrm{id}_p, d(m_p), \mathrm{relation})$ dans son propre corps signé, de sorte que

$$d(m_{\mathrm{enfant}}) \text{ engage } d(m_p) \text{ pour tout ancêtre } p, \tag{2}$$

comme un commit Git engage ses parents. Réécrire un ancêtre, même re-signé avec la clé légitime, change $d(m_p)$ et brise chaque descendant. Pour les jeux de données, l'arbre de Merkle (construction RFC 6962 [17] sur les hachages de lignes) dit *quelles* lignes ont changé, sous l'autorité de la racine signée ; les feuilles, dans un fichier annexe, ne sont crues que si elles reproduisent cette racine.

C. Vérification : dix contrôles, trois verdicts

La vérification d'un artefact exécute dix contrôles et descend récursivement dans ses parents (tableau II). Le verdict est dérivé, jamais fixé : ÉCHEC si un contrôle échoue, AVERTISSEMENT si l'intégrité tient mais qu'un contrôle de contenu a signalé les données ici ou en amont, VÉRIFIÉ sinon. VÉRIFIÉ signifie *exactement ce que des signataires de confiance, non révoqués et dans des rôles autorisés ont attesté, à travers une ascendance intacte* ; jamais « sûr ». La politique dit quel rôle signe quelle étape (Fig. 1) et quels parents chaque étape exige ou interdit : un modèle de base interne descend d'un jeu de données, un modèle affiné d'un jeu de données et d'exactement

TABLEAU I
CE QUI DISTINGUE AI-CHAINGUARD DES APPROCHES EXISTANTES (CAPACITÉS DOCUMENTÉES PAR CHAQUE PROJET, NOVEMBRE 2026 ; « NON » : HORS OBJECTIF). AUCUNE COLONNE N'ATTEINT « OUI » POUR LE CONTENU EMPOISONNÉ : C'EST LA FRONTIÈRE MESURÉE ICI.

	Sigstore / cosign	OpenSSF Model Signing	dispositions in-toto, provenance SLSA	Cartes de modèle, ML-BOM, registres (MLflow, DVC)	Scanners de portes dérobées	**AI-ChainGuard**
Périmètre	un artefact (image OCI, blob)	un répertoire de modèle	étapes d'une compilation	description, inventaire, versions	comportement du modèle	**jeu de données → base → affinage → déploiement**
Relie les étapes	non	non	la disposition relie les étapes	filiation par métadonnées	non	**oui : empreintes des parents dans chaque charge utile signée**
Qui signe quoi	politiques d'identité	identité du signataire	acteurs par étape ; plateforme (*builder*)	contrôle d'accès	s.o.	**rôle par étape, vérifié (V4, V8) ; éditeurs externes**
Révocation	expiration des certificats, journal	écosystème Sigstore	clé révoquée dans la disposition	transitions d'état	s.o.	**journal signé, ancré, en ajout seul ; propagée aux descendants**
Déploiement	contrôleurs d'admission	non	vérification du produit final	non	non	**passerelle : vérifier au déploiement, charger sûrement, tester, signer**
Contenu empoisonné	non	non	non	non	partiel, heuristique	partiel : heuristiques en AVERTISSEMENT seulement ; **mesuré, non revendiqué**

TABLEAU II
LES DIX CONTRÔLES D'UNE VÉRIFICATION

	Contrôle	Échoue quand
V0	Enveloppe	absente, mal formée ou d'un autre artefact
V1	Hachage du fichier	SHA-256 ou taille différents du corps signé
V2	Racine de Merkle	des lignes ont changé (le rapport les nomme)
V3	Signature Ed25519	invalide, ou clé absente du journal signé
V4	Confiance du signataire	clé révoquée, rôle ou identité discordants, rôle interdit pour l'étape, index et journal en désaccord
V5	Liens parentaux	parent sans enveloppe ou d'empreinte différente
V6	Ascendance	un ancêtre en ÉCHEC (récursif)
V7	Révocation	artefact ou ancêtre révoqué, ligne d'index supprimée ou drapeau effacé
V8	Politique	type incorrect pour l'étape, parents absents, interdits ou d'une étape ultérieure
V9	Drapeaux de contenu	contrôle de contenu signé déclenché ici ou en amont : AVERTISSEMENT seulement

un modèle de base, interne ou externe, et la clé d'entraînement ne peut signer l'importation externe (un modèle interne ne peut donc être réétiqueté « externe »). Chaque rapport consigne la tête du journal utilisée et nomme les artefacts externes dont la provenance s'arrête à un éditeur.

D. Le journal de confiance signé

Dans la version précédente (v0.3, octobre 2026), le magasin de confiance et les drapeaux de révocation vivaient dans l'index SQLite, et notre expérience de révocation en a montré la conséquence : quiconque pouvait écrire ce fichier pouvait effacer un drapeau `revoked`, et le modèle retiré était de nouveau vérifié. Confiance et révocation sont désormais les enregistrements d'un journal en ajout seul. L'enregistrement i porte un numéro de séquence, le hachage de l'enregistrement $i-1$, un horodatage, une action (`trust_key`, `revoke_key`, `revoke_artifact`, `rotate_root`) et son sujet, signés par la clé racine k_R du rôle `release_manager` :

$$r_i = \big(e_i,\ \mathrm{Sign}_{k_R}(\mathrm{canon}(e_i))\big), \qquad h_{i-1} \in e_i, \qquad (3)$$
$$h_i = \mathrm{SHA256}\big(\mathrm{canon}(r_i)\big).$$

Un fichier d'ancrage, dans le répertoire de clés, contient la clé publique racine et le couple (n, h_n) du dernier enregistrement. À chaque vérification, le journal est rejoué : chaque signature doit être valide sous la racine active (un `rotate_root` signé par l'ancienne racine en introduit une nouvelle), chaque enregistrement s'enchaîner au précédent, et le journal s'achever exactement à la tête ancrée. Journal tronqué (retour arrière), enregistrement modifié, journal supprimé ou substitué, clé présente dans l'index mais absente du journal : V3, V4 et V7 échouent en mode fermé, et un index en désaccord avec le journal est signalé comme altération. C'est un journal chaîné à rédacteur unique, assez petit pour se passer d'arbre de Merkle, avec l'ancrage de racine et la protection contre le retour arrière de TUF [16]. Pas de « dé-révocation » : un artefact retiré est remplacé par une nouvelle version.

E. Passerelle de déploiement et éditeurs externes

La passerelle est la dernière ligne de défense. Elle vérifie toute la chaîne au déploiement (une fois par déploiement, jamais par inférence), refuse un modèle en ÉCHEC et, par défaut, en AVERTISSEMENT, et refuse les formats pickle. Elle lit les octets du modèle une seule fois, les rehache et charge le modèle à partir de ces mêmes octets (pas de fenêtre TOCTOU), exécute un test de fumée sur des données tenues à l'écart, calcule une racine de publication (racine de Merkle des empreintes du modèle et de ses ancêtres) et signe un enregistrement de déploiement lié à la tête du journal utilisée. Un modèle d'un éditeur externe entre par un chemin distinct : l'éditeur signe un manifeste avec sa clé, un administrateur enregistre hors bande sa clé *publique*, et l'importation refuse tout ce qui est non signé ou mal formé, stocke le reste et le vérifie ; une importation en échec reste en quarantaine, ni parent ni déployable. Sa provenance s'arrête à l'éditeur.

F. Interopérabilité

Toute attestation s'exporte en énoncé in-toto v1 dans une enveloppe DSSE [14], [15], signée sur l'encodage de préauthentification (PAE) avec la clé de rôle du manifeste d'origine, avec un prédicat ChainGuard (le manifeste complet) ou SLSA Provenance v1 [9] : l'étape d'entraînement ou d'affinage est la compilation, jeu de données et modèle de base les dépendances résolues, le rôle signataire le *builder* ; empreinte et racine de Merkle voyagent comme sous-produits.

Un vérificateur DSSE indépendant valide l'export dans les tests, vecteur PAE de la spécification compris.

IV. Implémentation

AI-ChainGuard compte environ 8 800 lignes de Python 3.11+ (PyTorch, scikit-learn, FastAPI, Streamlit, bibliothèque `cryptography`) et 1 600 lignes de tests ; une bibliothèque centrale porte toute la logique de sécurité ; ligne de commande, API REST, pipeline, passerelle, scénarios d'attaque et intégration continue en sont de minces consommateurs. Clés privées : fichiers PKCS#8 à accès restreint ou secrets d'intégration continue ; index SQLite ; les enveloppes JSON sur disque font foi. Le pipeline de référence exporte le jeu *digits* de scikit-learn (images 8×8 ; 1 000 échantillons d'entraînement, 437 d'affinage, 360 tenus à l'écart), atteste les jeux de données avec des contrôles de contenu (distribution des étiquettes, doublons, déclencheur connu), entraîne un réseau convolutif de 38 282 paramètres, l'affine et le déploie par la passerelle. Un tableau de bord expose les mêmes opérations ; une pile Docker Compose et deux workflows GitHub Actions sont fournis. *Reproductibilité :* 168 tests en 36 s ; `attacks/run_all.py` régénère chaque tableau, `scripts/benchmark.py` chaque coût, et `paper/figures/make_figures.py` écrit dans `analysis.json` chaque nombre mesuré de la section V. Code, tests et scénarios sont publics (https://github.com/domail2karandeshmukh-cyber/ai-chainguard), et une page web vérifie et exécute le modèle signé de E2 dans le navigateur du lecteur (https://ai-chainguard.vercel.app).

V. Évaluation

A. Dispositif et protocole

Sauf mention contraire, les chiffres proviennent d'une exécution (graine 42) du 9 novembre 2026 dans l'environnement de référence, un conteneur Linux dans le cloud (deux cœurs logiques Intel Xeon à 2,10 GHz, 8 Go, sans GPU ; Python 3.13, PyTorch 2.15 sur CPU). Chaque variante part d'une copie fraîche d'une chaîne propre et attestée, applique une manipulation avec l'accès qu'aurait l'attaquant, vérifie l'artefact visé avec toute sa chaîne et consigne verdict, contrôles déclenchés et temps ; une variante est correcte si le verdict observé est celui prédit, et les témoins intacts doivent rester VÉRIFIÉ. Les expériences de porte dérobée entraînent de vrais modèles : le poison est un motif de coin 2×2 à la BadNets [13] sur 8 % des lignes d'entraînement, étiquette forcée à 7 ; le taux de succès de l'attaque (ASR) est la part des images de test des autres classes qui, porteuses du déclencheur, sont classées 7. Le pipeline conteneurisé, l'API et le tableau de bord ont aussi tourné sur un portable de l'équipe (MacBook Air) le 16 octobre 2026 ; les tests et le contrôle de publication tournent sur GitHub Actions (Python 3.11) à chaque push, et les coûts n'ont été mesurés que dans l'environnement de référence.

B. Les attaques après signature sont détectées

Le tableau III résume les sept expériences ; la Fig. 3(a) montre chaque variante. Les 48 variantes ont produit le verdict prédit ; toute manipulation postérieure à la signature a été rejetée, même ténue. Sur les jeux de données : étiquettes de cinq lignes inversées, une ligne ajoutée ou une seule espace finale (V1, V2, lignes nommées). Sur les modèles : un poids décalé de 10^{-6} ou des poids identiques réenregistrés dans un autre format (V1). Sur la provenance : enregistrement d'un parent supprimé, parent modifié re-signé avec la clé légitime, lien parental modifié sans re-signature, modèle sans ascendance de données (V5, V3, V8). Sur les clés : signatures corrompues ou rejouées, clé utilisée dans le mauvais rôle, clés inconnues ou révoquées (V3, V4). Révoquer un artefact ou une clé fait échouer chaque descendant et bloque le déploiement, tandis qu'un jeu de données signé par un autre rôle reste VÉRIFIÉ après la révocation d'une clé d'entraînement : le périmètre des dégâts est celui de la clé. La détection (vérification de l'artefact attaqué avec sa chaîne) a pris de 1,7 à 15,1 ms (médiane 7,5 ms).

TABLEAU III
LA SUITE D'ATTAQUES : 48 VARIANTES, TOUTES AVEC LE VERDICT PRÉDIT ; CONTRÔLES DÉCLENCHÉS PAR LES ATTAQUES DE L'EXPÉRIENCE.

Exp.	Expérience (scénario)	Var.	Verdict	Contrôles
1	Altération du jeu de données (A)	5	témoin VÉRIFIÉ ; 4 ÉCHEC	V1, V2, V6
2	Substitution du modèle (B)	4	témoin VÉRIFIÉ ; 3 ÉCHEC	V1
3	Signature invalide, abus de clé (D)	7	témoin VÉRIFIÉ ; 6 ÉCHEC	V3, V4, V6
4	Provenance brisée (C)	5	témoin VÉRIFIÉ ; 4 ÉCHEC	V3, V5, V8
5	Révocation, 8 contournements (F)	13	2 VÉRIFIÉ prévus ; 11 ÉCHEC, bloqués	V3, V4, V6, V7
6	Poison avant signature (E)	5	E0 témoin VÉRIFIÉ ; E1 ÉCHEC ; E3 AVERTISSEMENT ; E2, E4 VÉRIFIÉ	V6, V9
7	Source externe non fiable (G)	9	3 VÉRIFIÉ ; 4 ÉCHEC ; 2 REFUSÉ	V1, V3, V4, V6

C. Attaques contre la confiance elle-même

L'expérience 5 (scénario F) vise les mécanismes qui décident de la confiance : après la révocation d'un modèle de base, huit contournements, C1 à C8, tentent de faire vérifier de nouveau son descendant affiné. Éditer l'index (supprimer la ligne de l'artefact révoqué, C1 ; supprimer la clé révoquée, C2 ; écraser le statut en cache, C3) échouait déjà en v0.3 : le vérificateur ne lit jamais un statut en cache et traite une ligne absente comme une révocation possible (mode fermé). Effacer le drapeau `revoked` (C4), sa limite documentée, échoue désormais : le journal signé dit toujours « révoqué », et le désaccord lève un événement `INDEX_TAMPERING` (V6, V7). Une clé pirate écrite dans l'index avec le rôle d'entraînement et utilisée pour re-signer un modèle substitué (C5) échoue, car elle n'est jamais entrée dans le journal signé (V3, V4). Tronquer le journal (C6), modifier un enregistrement en place (C7) et supprimer le journal (C8) échouent en mode fermé (V3, V4, V6, V7) : le rejeu ne s'achève plus à la tête ancrée ou ne se vérifie plus sous la clé racine. Reste de confiance le répertoire de clés : qui détient la clé racine et l'ancre contrôle les décisions de confiance.

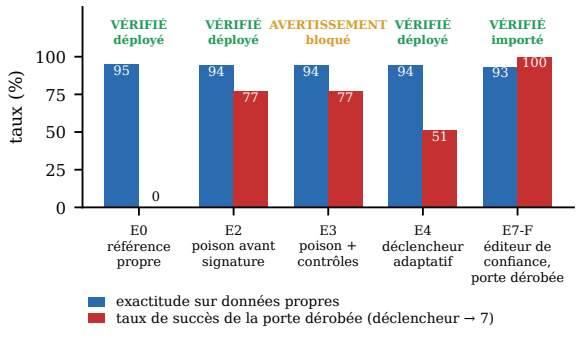


FIG. 2. Expériences 6 et 7 : exactitude propre et succès de la porte dérobée du modèle déployé ou importé, avec son statut. Le poison signé (E2) et la porte dérobée d'un éditeur de confiance (E7-F) passent tous les contrôles cryptographiques ; les contrôles de contenu attrapent un déclencheur connu (E3), pas un déclencheur adaptatif (E4).

D. Signé n'est pas sûr

Les expériences 6 et 7 (scénarios E et G) mesurent ce que la signature ne couvre pas (Fig. 2 ; E0 à E4 sont les variantes de l'expérience 6). Le pipeline propre (E0) déploie un modèle à 95,0 % d'exactitude et 0 % d'ASR. Du poison injecté dans le jeu de données *après* la signature (E1) fait échouer la chaîne en V6, comme attendu. Quand le responsable des données signe un jeu empoisonné *avant* l'attestation (E2), tout passe : le modèle affiné est déployé en VÉRIFIÉ avec 94,2 % d'exactitude propre, à moins d'un point du modèle propre, et 77,2 % d'ASR. Avec des contrôles de contenu antérieurs à l'attestation (E3), le déclencheur connu est signalé, le jeu de données porte un drapeau signé, le modèle passe en AVERTISSEMENT et la passerelle le bloque par défaut : les heuristiques valent comme indices. Un attaquant adaptatif qui déplace le déclencheur vers un motif inconnu des contrôles (E4) est déployé en VÉRIFIÉ avec 51,2 % d'ASR. Deux exécutions complémentaires du 10 novembre confirment la frontière. Répéter E0, E2, E3 et E4 avec dix autres graines ne change aucun statut : E2 est déployé avec 74 à 95 % d'ASR (médiane 83 %), E4 avec 58 à 90 % (médiane 70 %), et la graine de référence est en bas de plage. Empoisonner le jeu d'affinage (35 de ses 437 lignes) donne un déploiement VÉRIFIÉ avec 25 % d'ASR, et un AVERTISSEMENT avec les contrôles de contenu. L'expérience 7 répète la leçon à la frontière de la chaîne d'approvisionnement : un modèle à porte dérobée entraîné par un éditeur *de confiance*, enregistré (variante E7-F), est importé en VÉRIFIÉ avec 93,3 % d'exactitude propre et 100 % d'ASR, tandis qu'éditeur inconnu, modèle modifié en transit, modèle non signé et clé d'éditeur révoquée sont rejetés (tableau III). Une signature valide prouve *qui* a attesté et *que rien n'a changé depuis* ; elle transfère la confiance au signataire, elle n'en crée pas.

E. Coût

La Fig. 3(b) donne le coût. Attester les quatre artefacts du pipeline a pris 10,7 ms contre 1 288 ms d'entraînement (0,83 % ; part liée à la charge, l'attestation croissant avec les octets et l'entraînement avec le calcul). Une vérification de chaîne complète du modèle affiné et de ses trois ancêtres prend 8,5 ms (médiane de 200 exécutions ; p95 12,2 ms), dont 0,16 ms de rejeu du journal de 5 enregistrements (0,75 ms à froid). Le premier déploiement sécurisé coûte 253 ms, une fois : vérification de chaîne, écritures d'audit et lecture sûre prennent 31 ms, le chargement safetensors 1,1 ms et le test de fumée 8,3 ms ; les quelque 210 ms restants vont à la signature et à l'enregistrement du déploiement, qui vérifie la chaîne une seconde fois, et aux écritures d'événements et de métriques. Une prédiction coûte ensuite 0,084 ms avant et 0,085 ms après le déploiement vérifié : la vérification n'est pas sur le chemin de l'inférence. Les enveloppes signées font 1,4 à 3,1 Ko par artefact ; le fichier de Merkle d'un jeu de données est linéaire en ses lignes (30 et 68 Ko ici ; un million de lignes en 1,7 s). SHA-256 tourne à environ 1,2 Go/s (256 Mio en 220 ms) ; signature (0,04 ms) et vérification (0,12 à 0,14 ms) Ed25519 ne dépendent pas de la taille. Rejouer le journal coûte environ 0,14 ms par enregistrement avec chaque signature vérifiée (0,75 ; 6,9 ; 69 et 288 ms pour 5, 50, 500 et 2 000 enregistrements) et environ 15 μs par enregistrement une fois les signatures en cache ; au-delà de milliers d'enregistrements, un instantané signé remplacerait le rejeu depuis l'origine.

VI. DISCUSSION ET LIMITES

Les mesures sont une borne supérieure de ce que la conception offre. La suite d'attaques est écrite par les défenseurs et tourne sur un réseau convolutif pour chiffres 8×8 : elle montre que les contrôles cryptographiques se comportent comme prévu et l'ampleur du fossé sémantique sur un petit modèle, non sa taille pour un modèle de langue, où l'empoisonnement coûte moins [4]. Les contrôles de contenu sont des heuristiques simples, choisies pour montrer qu'elles se contournent (E4) ; des défenses plus fortes [21]–[24] relèveraient la barre sans la fermer. La confiance repose sur des clés Ed25519 de longue durée dans des fichiers : une clé volée et non révoquée signe des manifestes valides, d'où l'intérêt des certificats éphémères sans clé [11] et des clés matérielles. Le journal de confiance est local et à rédacteur unique : une copie périmée manquerait une révocation récente, et l'ancre est aussi critique que la clé racine. La forme canonique du JSON est propre à Python plutôt que conforme à la RFC 8785 [26], l'API n'a pas d'authentification, la passerelle vérifie la chaîne deux fois au premier déploiement, et les coûts n'ont été mesurés que dans un seul environnement. Rien de cela ne touche le résultat central, qui tient par construction : les signatures prouvent l'origine et l'intégrité, et un rapport qui affiche VÉRIFIÉ sans dire ce que ce statut couvre ou non invite à la mauvaise conclusion. AI-ChainGuard sépare donc preuves cryptographiques et de contenu (V0–V8 contre V9), étiquette la provenance externe, et ne laisse jamais une heuristique produire un ÉCHEC ni une signature un « sûr ».

VII. CONCLUSION

Nous avons construit une chaîne d'attestation pour pipelines d'IA qui relie jeux de données, modèles de base, modèles externes, modèles affinés et déploiements par des manifestes

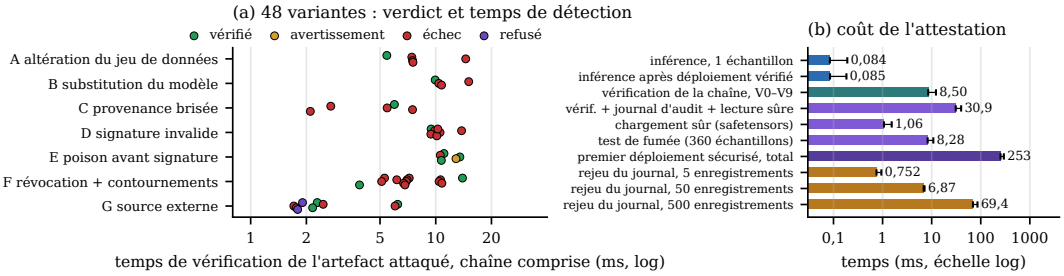


FIG. 3. (a) Verdict et temps de vérification des 48 variantes, par scénario (chaîne fraîche par variante ; le temps inclut l'ascendance et le rejeu du journal). (b) Coût de l'attestation (barre : médiane ; moustache : p95) : inférence avant et après déploiement vérifié, vérification de la chaîne, premier déploiement et ses composantes, rejeu de journaux de 5, 50 et 500 enregistrements.

signés contenant les empreintes de leurs parents, impose qui signe quelle étape, garde confiance et révocation dans un journal signé, ancré et en ajout seul, et conditionne le déploiement à la vérification de toute la chaîne. Sur 48 variantes, chaque manipulation postérieure à la signature a été détectée, contournements de l'index et du journal compris, pour moins d'un pour cent du temps d'entraînement, 8,5 ms par vérification et rien par inférence. La même suite montre la limite : un poison antérieur à la signature est déployé avec 77 % de succès de porte dérobée, la porte dérobée d'un éditeur de confiance avec 100 %. La sécurité des chaînes d'approvisionnement de l'IA a besoin des deux sortes de preuves, rapportées séparément ; ce travail fournit la première et mesure où la seconde doit prendre le relais. Prochaines étapes : publication du journal de confiance dans un journal de transparence partagé, signature sans clé, reproduction sur le matériel de l'équipe et sur des modèles plus grands.

REMERCIEMENTS

Les auteurs remercient Sofiane Hamrioui, ESAIP Angers, pour l'encadrement de ce projet. L'implémentation a été développée avec une génération de code assistée par IA et validée par exécution ; l'équipe répond de sa conception, de son évaluation et de ses conclusions.

RÉFÉRENCES

[1] Équipe PyTorch, « Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022 », 31 déc. 2022. https://pytorch.org/blog/compromised-nightly-dependency/ (consulté le 9 nov. 2026).

[2] D. Cohen, JFrog Security Research, « Data scientists targeted by malicious Hugging Face ML models with silent backdoor », févr. 2024. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/ (consulté le 9 nov. 2026).

[3] K. Zanki, ReversingLabs, « Malicious ML models discovered on Hugging Face platform », févr. 2025. https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face (consulté le 9 nov. 2026).

[4] A. Souly *et al.*, « Poisoning attacks on LLMs require a near-constant number of poison samples », arXiv:2510.07192 [cs.LG], oct. 2025.

[5] N. Carlini *et al.*, « Poisoning web-scale training datasets is practical », dans *Proc. IEEE Symp. Security and Privacy (SP)*, 2024, p. 407–425.

[6] Parlement européen et Conseil, « Règlement (UE) 2024/1689 du 13 juin 2024 établissant des règles harmonisées concernant l'intelligence artificielle (règlement sur l'intelligence artificielle) », *J. off. Union eur.*, L, 2024/1689, 12 juil. 2024, art. 15, par. 5.

[7] NIST, « Artificial Intelligence Risk Management Framework (AI RMF 1.0) », NIST AI 100-1, janv. 2023.

[8] MITRE, « ATLAS technique AML.T0010: AI supply chain compromise ». https://atlas.mitre.org/techniques/AML.T0010 (consulté le 9 nov. 2026).

[9] OpenSSF, « SLSA specification v1.1: Supply-chain Levels for Software Artifacts », avr. 2025. https://slsa.dev/spec/v1.1/ (consulté le 9 nov. 2026).

[10] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola et J. Cappos, « in-toto: Providing farm-to-table guarantees for bits and bytes », dans *Proc. 28th USENIX Security Symp.*, 2019, p. 1393–1410.

[11] Z. Newman, J. S. Meyers et S. Torres-Arias, « Sigstore: Software signing for everybody », dans *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2022, p. 2353–2367.

[12] OpenSSF, « Model Signing v1.0 », avr. 2025. https://github.com/sigstore/model-transparency (consulté le 9 nov. 2026).

[13] T. Gu, B. Dolan-Gavitt et S. Garg, « BadNets: Identifying vulnerabilities in the machine learning model supply chain », arXiv:1708.06733 [cs.CR], 2017.

[14] Projet in-toto, « in-toto Attestation Framework: Statement v1 ». https://github.com/in-toto/attestation (consulté le 9 nov. 2026).

[15] Secure Systems Lab, « DSSE: Dead Simple Signing Envelope, protocol v1 ». https://github.com/secure-systems-lab/dsse (consulté le 9 nov. 2026).

[16] J. Samuel, N. Mathewson, J. Cappos et R. Dingledine, « Survivable key compromise in software update systems », dans *Proc. ACM Conf. Computer and Communications Security (CCS)*, 2010, p. 61–72.

[17] B. Laurie, A. Langley et E. Kasper, « Certificate Transparency », RFC 6962, IETF, juin 2013.

[18] M. Mitchell *et al.*, « Model cards for model reporting », dans *Proc. Conf. Fairness, Accountability, and Transparency (FAT*)*, 2019, p. 220–229.

[19] T. Gebru *et al.*, « Datasheets for datasets », *Commun. ACM*, vol. 64, n° 12, p. 86–92, 2021.

[20] Hugging Face, « Safetensors: a simple, safe way to store and distribute tensors », documentation ; audit de sécurité par Trail of Bits, 2023. https://huggingface.co/docs/safetensors (consulté le 9 nov. 2026).

[21] B. Wang *et al.*, « Neural Cleanse: Identifying and mitigating backdoor attacks in neural networks », dans *Proc. IEEE Symp. Security and Privacy (SP)*, 2019, p. 707–723.

[22] Y. Gao, C. Xu, D. Wang, S. Chen, D. C. Ranasinghe et S. Nepal, « STRIP: A defence against trojan attacks on deep neural networks », dans *Proc. Annu. Computer Security Applications Conf. (ACSAC)*, 2019, p. 113–125.

[23] B. Tran, J. Li et A. Madry, « Spectral signatures in backdoor attacks », dans *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018.

[24] B. Chen *et al.*, « Detecting backdoor attacks on deep neural networks by activation clustering », dans *Proc. AAAI Workshop on Artificial Intelligence Safety (SafeAI)*, 2019.

[25] S. Josefsson et I. Liusvaara, « Edwards-Curve Digital Signature Algorithm (EdDSA) », RFC 8032, IETF, janv. 2017.

[26] A. Rundgren, B. Jordan et S. Erdtman, « JSON Canonicalization Scheme (JCS) », RFC 8785, IETF, juin 2020.