



Pipeline d'analyse automatique (multi-LLM) d'un code Java/Android

L'objectif est de combiner **analyse statique approfondie** et **IA générative** pour extraire un maximum d'informations utiles du code décompilé (ex. APK d'Android). On utilise plusieurs modèles LLM spécialisés (un pour l'analyse, un pour la validation), ainsi que des fonctions de recherche/embeddings et d'extraction de code. Cette approche *multi-LLM* permet de tirer parti des forces de chaque modèle (raisonnement avancé, rapidité, spécialisation code) pour une analyse fiable et exhaustive ¹.



Étape [0] – Extraction et préparation du code

- **Rôle** : Obtenir le code source sous une forme exploitable.
- **Actions** : Décompilation de l'APK (avec Jadx) en fichiers `.java` et export au format JSON pour faciliter l'analyse structurelle (AST).
- **Sortie** : Arborescence de classes Java avec signatures (nom, modificateurs, héritage), corps de méthodes, etc. Ces données peuvent être chargées dans des structures Python (dictionnaires, objets AST) pour le traitement.
- **Modèle** : Aucun LLM ici (opération déterministe). Outils suggérés : **Jadx**, **javalang** ou **JavaParser** pour construire l'AST.

Étape [1] - Analyse statique structurelle (AST)

- **Rôle** : Construire une base de faits complète sur le code (structure de classes, call graph, dépendances).
- **Entrée** : JSON / fichiers `.java` issus de l'étape précédente.
- **Actions** : Parser chaque fichier pour extraire :

 - Signatures de classes (modificateurs, interfaces implémentées) et champs constants.
 - Signatures de méthodes (types de retour, visibilité, paramètres).
 - Corps de méthodes (opérations de haut niveau, appels internes).
 - Graphe d'appels entre méthodes et classes.

- **Sortie** : Base de données ou objet en mémoire contenant la structure du code : liste de classes, de méthodes, de dépendances inter-classes, etc.
- **Modèle** : Aucun LLM. Utiliser **outils de parsing Java** (par exemple `javalang` ou `Eclipse JDT`). Cette étape prépare les métadonnées pour les analyses suivantes.

Étape [2] - Indexation sémantique (embeddings)

- **Rôle** : Permettre la recherche intelligente dans le code pour fournir un contexte pertinent aux LLMs.
- **Entrée** : Chunks de code (classes ou méthodes) extraits à l'étape 1.
- **Actions** :

 - Fragmenter le code en segments cohérents (par classes, méthodes ou blocs AST) pour respecter les limites des modèles.
 - Générer des *embeddings* vectoriels pour chaque segment à l'aide d'un modèle d'embeddings adapté (par exemple un modèle pré-entraîné sur du code ou texte, voir **CodeBERT**, **all-MiniLM**, etc.).
 - Indexer ces vecteurs dans une base de données vectorielle (FAISS, Qdrant, etc.).

- **Sortie** : Index de recherche sémantique: pour toute requête textuelle ou segment de code, on peut retrouver rapidement des morceaux de code similaires sémantiquement. Ce mécanisme RAG (« Retrieval-Augmented Generation ») permet de limiter le contexte envoyé au LLM à l'information la plus pertinente ².
- **Modèle** : Modèle d'embeddings. Par exemple **code-search-embedding** (CodeBERT) ou un petit modèle général (comme `all-MiniLM-L6-v2`) utilisé sur des représentations synthétiques du code ². Ces modèles sont généralement plus légers et rapides qu'un LLM, favorisant la vitesse.

Étape [3] - Analyse statique initiale (non-LLM)

- **Rôle** : Rechercher automatiquement certains patterns connus (sécurité, qualité, crypto) avant d'impliquer le LLM.
- **Actions** :
- **Recherche de crypto-API** : Parcourir le code pour détecter l'utilisation de classes cryptographiques (ex. `MessageDigest`, `Cipher`, `KeyPairGenerator`, etc.), algorithmes (MD5, SHA-1, AES, RSA, ECB, etc.), tailles de clés, modes d'opération, etc.
- **Détection de patterns dangereux** : Par exemple, chaînes codées en dur (clés, URI sensibles), accès au réseau, appel à des fonctions natives (JNI), utilisation de `Runtime.exec`, de la réflexion (`Class.forName`), stockage local de données sensibles, etc.
- **Vérification basique de permission/policy** : (dans le code, on peut relever des indicateurs d'accès restreint ou de vérification de signature).

- **Compilation des résultats** : Liste de warnings et points critiques pour guider l'analyse LLM (ex. méthode `decryptPassword()` utilise SHA-1).
- **Sortie** : Rapport de pré-analyse avec findings (similaire à un SAST rudimentaire). Cela alimente les prompts du LLM pour focaliser sur les éléments sensibles.
- **Modèle** : Principalement des règles/SAST classiques. (Outils possibles : **FindSecBugs**, **Semgrep**, ou scripts Python basés sur AST.) Aucune IA ici, c'est un filtrage rapide pour aiguiller le LLM.

Étape [4] – Analyse macro (LLM-A)

- **Rôle** : Comprendre le **contexte global** et le rôle de chaque classe critique dans l'application.
- **Entrée** : Métadonnées extraites (nom de classe, package, hiérarchie, méthodes clés) et résumé de l'analyse initiale (étape 3).
- **Demande type (prompt)** : « À partir des informations suivantes sur une classe Java (signatures, noms, constantes), explique : le rôle général de la classe, son importance dans l'application, le contexte probable d'utilisation. »
- **Sortie** : Pour chaque classe analysée : description haut-niveau, indication de criticité, hypothèses sur la fonctionnalité. Par exemple, « `ApkSigningBlockUtils` semble gérer la vérification cryptographique des APK ; c'est donc critique pour la sécurité » ³.
- **Modèle (LLM-A)** : Un LLM volumineux et efficace sur du code. Par exemple, **CodeLlama 13B** (spécialisé code) ou **Llama 3.1 8B** pour un bon compromis qualité/vitesse ⁴. On privilégie un modèle qui gère bien le raisonnement sur code et le français (le prompt peut être en français ou anglais).

Étape [5] – Analyse structurelle et grouping (LLM-A)

- **Rôle** : Découper le code en sous-systèmes ou groupes fonctionnels internes, et identifier les méthodes critiques.
- **Entrée** : Liste des méthodes de la classe (signatures, visibilités, retours). Contexte issu de la macro-analyse (étape 4).
- **Actions** : Le LLM regroupe mentalement les méthodes selon leur responsabilité (ex. « cryptographie », « gestion binaire du fichier APK », « I/O », « utils »).
- **Demande type** : « Regroupe les méthodes suivantes par domaine fonctionnel (cryptographie, parsing, utilitaires, etc.), explique le rôle de chaque groupe et ses dépendances clés. »
- **Sortie** : Schéma de l'architecture interne de la classe : listes de méthodes par catégorie, rôle de chaque catégorie. Identification des méthodes centrales par groupe.
- **Modèle** : Toujours LLM-A (CodeLlama ou similaire). On peut, si utile, fournir le corps d'une méthode clé pour éclairer certains regroupements (via appel de fonction).

Étape [6] – Boucle d'exploration dynamique (LLM-A + embeddings + fonction-calling)

- **Rôle** : Lever les zones d'ombre : lorsqu'une méthode référencée n'est pas encore analysée, explorer dynamiquement les dépendances **uniquement si nécessaire**.
- **Mécanisme** :
- **Détection de besoin** : LLM-A, en analysant un flux d'exécution hypothétique, peut « repérer » des appels vers des méthodes externes ou classes non encore traitées. Il signale alors qu'il lui manque des informations.
- **Recherche avec embeddings** : Pour affiner la recherche, on utilise la recherche sémantique : par exemple, le LLM-A peut générer une requête textuelle (type « compute digest on APK ») et interroger l'index d'embeddings pour trouver des méthodes/classe possiblement liées.

- **Appels de fonction vers l'environnement Python :** Utilisation du mécanisme *function-calling*
5 . On expose des fonctions comme :

```
get_method_body("ClassName", "methodName")
get_called_methods("ClassName", "methodName")
get_callers("ClassName", "methodName")
analyze_class("OtherClassName")
semantic_search("recherche texte ou code")
```

LLM-A peut ainsi demander explicitement au programme d'extraire le corps d'une méthode ou d'enquêter sur des dépendances.

- **Mise à jour itérative :** Après chaque appel, on enrichit le contexte du LLM (nouveaux segments de code analysés) et on reprend l'analyse.
- **Condition de boucle :**

Tant que l'LLM-A identifie des méthodes/classes nécessaires non analysées, on récupère ces informations et on réitère 5 .

- **Exemple :** LLM-A note que `verifyIntegrity()` appelle `computeApkVerityDigest` (classe `DigestUtils` inconnue). Il invoque `get_method_body("DigestUtils", "computeApkVerityDigest")` pour obtenir son code, puis intègre cette méthode à son analyse. Ensuite il peut détecter que `DataSource.read()` est utilisée, et demander `analyze_class("DataSource")`, etc.
- **But :** Ne pas saturer le LLM avec tout le projet, mais explorer **ciblé**. Grâce à la recherche sémantique, on priorise les classes réellement liées (vs un balayage exhaustif).

Étape [7] - Analyse spécifique des primitives cryptographiques (LLM-A)

- **Rôle :** Examiner en détail toute la logique cryptographique du code.
- **Entrée :** Code des méthodes utilisant la crypto (extrait des étapes 1 et 6), plus métadonnées (algorithmes détectés à l'étape 3).
- **Actions :**
 - LLM-A repère les algorithmes (Symétriques, Asymétriques, hash, signatures) et vérifie leurs usages (par ex. mode AES, padding RSA).
 - Pour chaque primitive, il analyse si les paramètres sont adéquats (taille de clé suffisant, vecteur d'initialisation aléatoire correct, entropie, salage, etc.).
 - Il compare aux bonnes pratiques (ex. éviter MD5/SHA-1, préférer SHA-256 ; pas d'ECB sans IV, etc.) et signale les points faibles potentiels.
 - Il détecte l'usage de générateurs aléatoires cryptographiquement sûrs (`SecureRandom`) vs non-sécurisés.
- **Sortie :** Liste de constats et recommandations crypto : primitives utilisées (avec leurs risques), conseils d'amélioration. Ex. « Le code utilise MD5 pour l'intégrité – vulnérable aux collisions » 6 .
- **Modèle :** LLM-A (le même, avec la connaissance du domaine crypto). L'LLM peut être amené à faire plusieurs passes (one prompt per primitive). On peut structurer le prompt: « Pour chaque méthode de cette classe utilisant la cryptographie, donne l'algorithme, son usage et les risques éventuels. ».

Étape [8] - Analyse sécurité et détection de vulnérabilités (LLM-A)

- **Rôle** : Identifier tout comportement susceptible de compromettre la sécurité ou de générer un bogue critique.
- **Entrée** : Contexte complet obtenu (flux d'appel probable, données sensibles, résultats des étapes précédentes).
- **Actions** :
- Le LLM-A inspecte le *data flow*: comment les données entrent et sortent, vérifie les contrôles d'accès (par ex. appels de méthodes restreintes).
- Il recherche des vulnérabilités standards : injection de commande, détournement de serialization, débordements dans le parsing binaire, mauvaise validation d'entrées, etc.
- Il recoupe avec les résultats SAST initiaux pour confirmer ou infirmer les alertes (grâce à sa capacité de « raisonnement contextuel » ⁶).
- Il hiérarchise les risques (impact pour l'utilisateur / système si exploité).
- **Sortie** : Rapport de sécurité : liste de failles potentielles, méthodes critiques, données sensibles en jeu, niveau de gravité, recommandations. Par exemple : « Si `verifyIntegrity` est contourné, un APK malveillant pourrait être installé ³ ».
- **Modèle** : LLM-A, possiblement avec un jeu de prompts spécialisé « sécurité ». On peut inclure des *system prompts* style « You are a security auditor... ». La citation [9†L688-L692] confirme l'intérêt de LLMs pour détecter précocement les vulnérabilités.

Étape [9] - Validation indépendante (LLM-B)

- **Rôle** : Un second modèle (différent de LLM-A) relit et critique l'analyse produite pour détecter erreurs ou oubli.
- **Entrée** : Synthèse des analyses précédentes (macro, structurelle, crypto, sécurité).
- **Actions** :
- LLM-B, configuré en “auditeur/code reviewer”, vérifie la cohérence logique (pas de contradiction), relève les hypothèses non étayées ou risques sous-estimés.
- Il peut souligner ce qui manque (ex. « Et si un attaquant injecte des données ici ? »), et proposer des nuances.
- **Sortie** : Retour de validation : corrections, ajouts, mention du niveau de confiance global. Ce feedback est intégré au rapport final.
- **Modèle (LLM-B)** : Un LLM différent (p.ex. **Mistral 7B** ou **Llama 3.1 8B**). On cherche un compromis vitesse/confiance : Mistral est très rapide et puissant ⁴. Il doit aussi être compétent en code. On lui donne un prompt type « Tu es un auditeur indépendant... vérifier ce qui précède... ».

Étape [10] – Synthèse et rapport final

- **Rôle** : Produire un rapport structuré et lisible à partir de toutes les analyses.
- **Actions** :
- Générer un document final (Markdown, PDF, etc.) incluant : résumé exécutif, description générale de l'application, architecture, liste des classes/méthodes critiques, points forts/faibles, vulnérabilités identifiées, recommandations de sécurité, scores ou indicateurs de confiance.
- Classer les findings (par classe ou par type de risque).
- Éventuellement utiliser un LLM (avec prompts orientés documentation) pour formater élégamment le résultat.
- **Sortie** : Rapport complet d'analyse, prêt à être lu par un ingénieur/relecteur.

Modèles et outils recommandés

- **LLM principal (LLM-A)** : Un modèle **code-spécialisé** tel que CodeLlama 13B (performance excellente sur le code ⁴) ou Llama 3.1 8B.
 - **LLM d'audit (LLM-B)** : Un second modèle, par exemple Mistral 7B (très rapide) ou un autre Llama (différent de LLM-A).
 - **Embeddings** : Modèle d'embeddings de code (p.ex. CodeBERT / `all-MiniLM-L6-v2` avec représentation de code).
 - **Fonctions externes** : Implémentées en Python, par exemple à travers *LangChain* ou OpenAI function-calling API.
 - **Autres outils** : Analyse statique SAST (FindSecBugs, Semgrep), générateurs de code (Langchain, tree-sitter pour splitting AST), tests unitaires automatiques (pour valider certaines hypothèses), etc.
-

Illustration du flux d'exécution : L'analyse s'effectue de manière itérative. Par exemple, pour `ApkSigningBlockUtils`, on commence par décrire la classe et ses méthodes (étapes 4-5), puis on voit que `verifyIntegrity()` appelle une méthode de signature externe. Grâce aux embeddings, on retrouve des méthodes similaires de calcul de digest, puis on appelle dynamiquement `analyze_class("DataSource")` via *function-calling* pour comprendre le flux binaire. Cette boucle continue jusqu'à ce que le modèle ait suffisamment d'informations. Enfin, on liste les risques (p.ex. vérification manquante, cryptographie faible) et on fait valider par LLM-B ⁶ ³.

Conclusion : Cette pipeline non-linéaire combine extraction AST, recherche sémantique, et raisonnement LLM pour couvrir : - **Structure & fonctionnalité** (groupe de méthodes, rôle des classes)
- **Cryptographie** (algorithmes utilisés, conformité aux standards)
- **Sécurité** (vulnérabilités potentielles, données sensibles)
- **Itérations ciblées** (exploration dynamique guidée par LLM)
- **Validation croisée** (audit par second LLM) ¹ ⁶.

L'utilisation d'un modèle spécialisé code (p.ex. CodeLlama) et d'un modèle rapide pour l'audit améliore à la fois la **précision** et la **vitesse** de l'analyse ⁴. Enfin, l'usage d'**embeddings et de fonction-calling** permet d'éviter la surcharge de contexte et de se concentrer sur le code vraiment pertinent ² ⁵.

¹ Multi-LLM Debugging Workflow Guide | by Oscar | Medium
<https://medium.com/@dev-Oscar-checklive/multi-llm-debugging-workflow-guide-e6df0cdc0747>

² Semantic Code Search. There's been a lot of buzz lately about... | by Xiaojing | Medium
<https://medium.com/@wangxj03/semantic-code-search-010c22e7d267>

³ Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets
<https://arxiv.org/html/2503.17502v1>

⁴ Llama 3.2 vs Mistral 7B vs CodeLlama: Which Wins? (Tested) | Local AI Master
<https://localaimaster.com/blog/llama-vs-mistral-vs-codellama>

⁵ Function calling using LLMs
<https://martinfowler.com/articles/function-call-LLM.html>

⁶ Using LLMs to filter out false positives from static code analysis | Datadog
<https://www.datadoghq.com/blog/using-llms-to-filter-out-false-positives/>