SENG 696 – Report 1 (GAIA)

Preliminary Design: Multi-Agent Security-Risk Triage for C/C++ Crypto Code

Reyhaneh Farahmand
UCID: 30271200

October 3, 2025

Submission

- **Project Repo:** https://github.com/Rey1380/SENG696-Reyhaneh-Farahmand-Report-1

1 System Scope & Motivation

We design a **Multi-Agent System (MAS)** that automatically generates an **actionable security-risk report** for C/C++ cryptographic code (`AES`, `SHA`, `ECDSA`, etc.).

The MAS fuses:

1. Per-file likelihood of being *AI-generated*;
2. Static-analysis vulnerability findings;
3. Then computes composite risk-scores and ranks findings for remediation.

Boundary: Input is a local repo or PR-diff. The system runs a Python AI-origin detector plus a static scanner (e.g. `flawfinder`/`cppcheck`) and outputs a structured JSON plus human-readable Markdown/PDF report.

Assumptions & Constraints: C/C++ only; MVP uses a single scanner; No network calls; Read-only analysis.

2 Choice of Methodology

GAIA focuses on:

- **Role Model**: Responsibilities (liveness & safety), permissions, protocols;
- **Interaction Model**: Recurring inter-role protocol patterns;

- **Design Models**: Agent, service, and acquaintance diagrams mapping roles to implementable agents.

This organizational perspective matches our 4-agent MAS and can be implemented in JADE or similar. Other AOSE methods such as MaSE or Tropos were reviewed but GAIA was chosen for its role/protocol clarity and direct fit to our problem.

3 GAIA Analysis – Roles Model

GAIA sees the system as a *society of interacting roles*. Below are our four roles with key responsibilities (split into *Liveness* – what must eventually be achieved – and *Safety* – what must always hold), plus required permissions and protocols.

Roles and Responsibilities

Orchestrator:

Liveness: (i) On `AnalyzeRepo` request, sequentially trigger `AnalyzeOrigin` `→ScanVulns→Triage`; (ii) Gather results; (iii) Output consolidated report.
Safety: Read-only repo access; fail-safe if any sub-task fails.
Permissions: Read repo/config; write final JSON & PDF.
Protocols initiated: AnalyzeOrigin, ScanVulns, Triage.

AI-Origin:

Liveness: On AnalyzeOrigin, return per-file $p_{AI} \in [0, 1]$.
Safety: Sandboxed ML inference; no network.
Permissions: Read source files; write `origin.json`.
Protocols responded: AnalyzeOrigin.

Vulnerability:

Liveness: On ScanVulns, run static scanner and return normalized findings.
Safety: Scanner whitelisted; no repo modification.
Permissions: Read source files; write `vuln.json`.
Protocols responded: ScanVulns.

Severity:

Liveness: On Triage, fuse severities with p_{AI} to compute risk ranking.
Safety: Deterministic reproducible scoring; no external calls.
Permissions: Read `origin.json`, `vuln.json`; write `triage.json`.
Protocols responded: Triage.

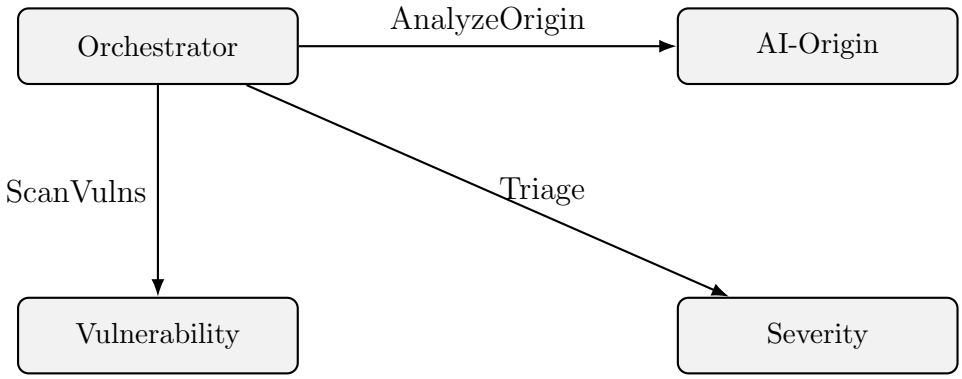

Figure 1: GAIA Roles Model – Key roles and protocol directions.

4 GAIA Analysis – Interaction Model

Each protocol documents *Purpose, Initiator, Responder, Inputs, Outputs*.

- **AnalyzeOrigin:** Orchestrator→AI-Origin; In: repo-path → Out: `origin.json`.
- **ScanVulns:** Orchestrator→Vulnerability; In: repo-path, tool → Out: `vuln.json`.
- **Triage:** Orchestrator→Severity; In: `origin.json`, `vuln.json` → Out: `triage.json`.

Main Scenario – Clean Sequence Diagram

5 GAIA Design Models

- **Agent Model:** One-to-one mapping of roles to OrchestratorAgent, AiOriginAgent, VulnAgent, SeverityAgent.
- **Service Model (Examples):**
  - OrchestratorAgent: AnalyzeRepo(), RequestOrigin(), RequestVulns(), RequestTriage()
  - AiOriginAgent: ComputeOrigin()
  - VulnAgent: RunScanner()
  - SeverityAgent: FuseAndRank()
- **Acquaintance Model:** Directed links Orchestrator→(AI-Origin,Vuln,Severity); Reverse links for responses.

6 Risk Scoring Policy

Severity levels: LOW = 0.30, MED = 0.60, HIGH = 0.85, CRITICAL = 1.00. For a vulnerability of base-severity s in a file with AI-origin p_{AI}:

$$risk \;=\; \min\bigl(1,\; s \times (1 + 0.20 \times p_{AI})\bigr)$$

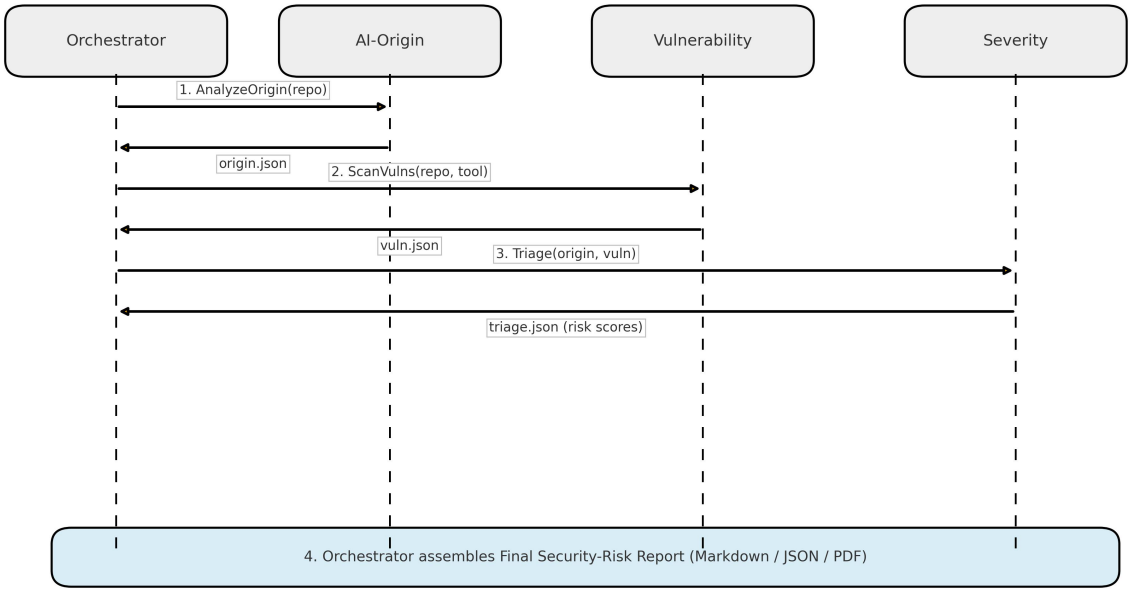

Figure 2: GAIA Interaction Model.

As part of our proposed risk–scoring approach, we incorporate the probability that a code fragment was generated by an AI model as an additional factor that can *amplify the baseline severity* reported by the static–analysis tool. The underlying assumption is that **AI–generated code may carry a higher security risk** compared to code written and reviewed entirely by experienced human developers. This elevated risk arises from factors such as the model's limited understanding of the full system context, the possibility of reusing outdated or unsafe coding patterns, and the likelihood that developers may integrate AI–suggested code without performing thorough manual review or security validation.

By including the AI–origin likelihood p_{AI} in the formula, our intention is to adjust the prioritization of findings: vulnerabilities detected in predominantly AI–authored code receive proportionally higher risk scores and are surfaced earlier in the remediation queue. The multiplier constant (e.g., 0.20 in our prototype) reflects a *policy assumption* about the magnitude of this additional risk. This parameter is not fixed; it can be increased, decreased, or even replaced by a non–linear function as we gather empirical evidence or consult with domain–specific security experts. This design choice keeps the framework flexible and extensible, enabling security teams to calibrate the risk–amplification factor to align with their organizational threat models and the evolving understanding of AI–generated code quality.

In addition, we apply an *optional file–level aggregation step* to compute an overall risk score for each source file. The formula

$$\text{FileRisk} = 1 - \prod_{i \in findings(file)} (1 - risk_i)$$

captures the probability that at least one detected vulnerability in the file poses a security

risk. This method ensures that multiple low–severity findings can combine into a higher file–level risk, while keeping the result bounded between 0 and 1. It provides a principled way to prioritize files with a dense concentration of issues, even if each individual vulnerability is minor.

7 Planned Implementation Milestones

1. Finalize GAIA analysis & design models (this report)
2. Implement JADE agent skeletons + JSON messaging
3. Integrate Python AI-Origin detector and scanner adapter
4. Add Severity fusion logic + Markdown/PDF report generator
5. Test MAS on open-source C/C++ crypto repos