Network Working Group
Internet-Draft
Intended status: Informational
Expires: 29 October 2026

G. Kayas
J. Sampathkumar
B. Sekar
S. S. G. B
J. Dev
Comcast Corporation

27 April 2026

FLAIR: Framework for Language-Agnostic Discovery of Cryptographic Elements using Semantic Graphs

draft-comcast-flair-crypto-discovery-00

Abstract

This document introduces FLAIR (Framework for Language-Agnostic Intermediate Representation), a novel approach to discover cryptographic elements in source code. By detecting encryption components present in code, FLAIR addresses critical challenges like identifying deprecated and misconfigured implementations across diverse programming languages. Unlike language-specific tools, FLAIR employs semantic graphs to represent instances of encryption detected, enabling identification of cryptographic algorithm usage, associated parameters (keys, nonces, salts), and flag compliance status with established encryption standards. The semantic graph approach makes representing cryptographic algorithms and related elements independent of the underlying programming language.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 29 October 2026.

Table of Contents

1. Problem Statement

Modern enterprise and cloud infrastructure increasingly relies on encrypted communication to protect sensitive data in transit. With the increasing integration of encrypted communication across enterprise and cloud infrastructure, a growing number of network outages and performance degradations are now traced to undetected cryptographic vulnerabilities in code [ref1]. Organizations face growing security risks from:

1. Deprecated cryptographic algorithms that no longer meet security standards

2. Misconfigured cryptographic implementations causing network outages

3. Incompatible cryptographic implementations across heterogeneous systems

4. Lack of visibility into cryptographic usage across enterprise codebases that impacts migration efforts to newer encryption standards, like Post-Quantum Cryptography (PQC)

5. Achieving comprehensive coverage without developing language-specific solutions

However, the proliferation of diverse programming languages, cryptographic libraries, and implementation patterns in source code creates significant impediment in building a holistic visibility tool capable of detecting and analyzing cryptographic usage across diverse network applications. Consequentially, organizations may struggle to identify and remediate cryptographic vulnerabilities before they lead to service disruptions.

2. Alternate Approaches to Cryptographic Discovery in Code

Current open-source cryptographic detection solutions fall into two categories with distinct limitations: (1) general-purpose static analyzers (e.g., Bandit, Snyk, CodeQL, Joern, etc.), and (2) language-specific tools (e.g., CryptoGuard, CogniCrypt, Cryptolation, etc.).

2.1. General-Purpose Static Analyzers

Tools such as Bandit, Snyk, CodeQL, and Joern/CPG [ref2] are examples of general purpose static analyzers. They are typically used to detect vulnerabilities in code, such as security bugs, vulnerable libraries, and policy violations before the code is deployed. Static code analyzers have wide language coverage - there is a tool for nearly all major programming languages today. Cryptographic discovery is possible using these tools. They can break down code into graphs that can follow rules to detect specific cryptographic usage. However, such detections can have a high false-positive rate and will need several rules across different languages to achieve traceability of cryptographic materials (e.g., keys, nonces, salts, digests).

2.2. Language-Specific Tools

Tools such as CryptoGuard, CogniCrypt, and Cryptolation are examples of language-specific tools. These open-source tools provide precision within supported languages. In many cases, they go beyond pattern-matching to provide semantic understanding of cryptographic usage using machine learning models. However, they are designed to work extremely well for a single programming language. Enterprises which have polyglot codebases suffer from limited visibility during cryptographic discovery. Even though they receive high precision results from these tools, they would have to maintain multiple toolchains to cover their entire codebase. This leads to significant development and maintenance overhead.

3. Definitions

Semantic graphs:
: Graphs that represent the semantic meaning of code elements and their relationships, enabling deeper understanding of program structure and behavior.

Code property graphs:
: Graphs that represent code elements and their properties, including control flow, data flow, and inter-element relationships.

Cryptographic elements:
: Components involved in cryptographic operations, such as algorithms, keys, nonces, salts, and message digests.

Language-agnostic:
: An approach that is independent of any specific programming language, allowing for broad applicability across diverse codebases.

Polyglot codebases:
: Codebases that contain source code written in multiple programming languages.

Deprecated cryptographic algorithms:
: Cryptographic algorithms that are no longer considered secure or recommended for use, such as MD5 and SHA1.

Misconfigured cryptographic implementations:
: Cryptographic implementations that do not adhere to best practices or security standards, leading to vulnerabilities.

NIST
: National Institute of Standards and Technology, a U.S. federal agency that develops and promotes measurement standards, including cryptographic standards.

Post-Quantum Cryptography (PQC):
: Cryptographic algorithms designed to be secure against attacks from quantum computers, such as lattice-based or hash-based algorithms which have been standardized by NIST.

Alias analysis:
: A technique used in program analysis to determine if two expressions in a program may refer to the same memory location.

4. Proposed Solution: FLAIR Framework

This draft proposes a new framework for cryptographic discovery that combines the best of both existing approaches - language-specific tools with the coverage of general purpose static analyzers. There is a critical need for a language-agnostic approach which can (1) identify the usage of cryptographic libraries, (2) determine cryptographic parameters supporting their usage (keys, nonces, salts), and (3) specify which cryptographic algorithms are being invoked. This draft proposes a novel semantic approach that detects cryptographic libraries, how the invoked algorithms methods in these libraries get accessed throughout the code, and the associated cryptographic parameters. This language-agnostic approach overcomes the limitations of existing solutions by producing a unified JSON representation across diverse languages (Python, Java, JavaScript, etc.) to support cross-language rule development.

4.1. Design Principles

FLAIR is the first cryptographic detection tool achieving language independence through semantic graphs. This approach makes cryptographic discovery independent of function nomenclature and variable naming conventions. The detection of relationships between libraries, keys, and function calls remains unaffected by variable renaming or library substitution. Semantic graphs have demonstrated effectiveness in biological sciences [ref3]. This approach reduces false positives compared to pattern-matching solutions. Additionally, FLAIR can detect keys propagated across multiple function calls throughout an entire codebase.

FLAIR also detects and stores cryptography-relevant information only. By limiting crypto-relevant nodes and edges, FLAIR remains lightweight even on large codebases while achieving efficiency.

4.2. Technical Components and Architecture

FLAIR consists of two main components:

1. Feature Extraction and Normalization: This component processes source code from multiple programming languages to extract cryptography-relevant information and normalize it into a unified intermediate representation (IR).

2. Crypto Semantic Graph (CSG) Construction and Backtracking: This component constructs a Crypto Semantic Graph (CSG) from the normalized IR, enabling multi-hop backtracking to identify relationships between cryptographic elements and their usage in code.

The following diagram illustrates the overall FLAIR technical flow and components.

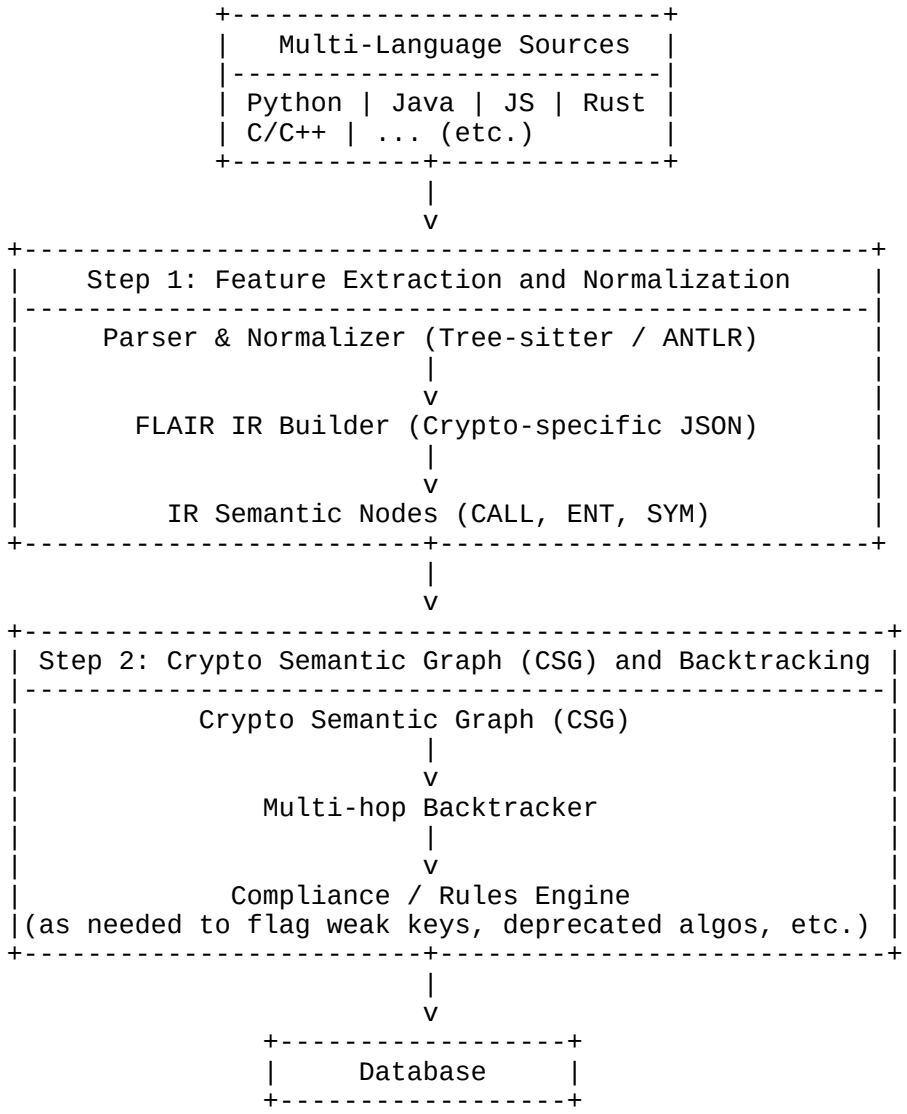

Figure 1: FLAIR process flow diagram.

5. Technical Details

5.1. Step 1: Feature Extraction and Normalization

The first step utilizes a general purpose parser (like Tree-sitter or ANTLR) to parse the various cryptographic libraries and code constructs. The language-specific components are then generalized to their core algorithms and their specifications (like "ECDSA", "RSA", with parameters like key sizes, e.g., "256"). The resultant JSON from the parser is then converted into a FLAIR IR (Intermediate Representation) graph JSON. This resultant IR now contains only mappings between detected libraries and their usage in code. The graph structure utilizes nodes and edges - Nodes, to represent cryptographic information with contextual metadata; and Edges, to represent relationships between information and its code usage.

Four sub-processes execute sequentially in this step:

1. Library Detection: Identifies cryptographic libraries through import statements and source references (e.g., OpenSSL). FLAIR uses partial parsers (e.g. Tree-sitter or ANTLR grammars) for each language. The adapter walks the AST to find calls, imports, and literal constructions. Only crypto-relevant code is analyzed.

2. Normalized Crypto API Call Detections: Detects API calls to cryptographic functions, normalizing them to primitive representations (e.g., CryptoJS.AES.encrypt() becomes AES.encrypt). A central table maps specific API patterns or strings to normalized names. E.g. all patterns matching cryptography.*.ec.ECDSA normalize to ECDSA; an arg "SHA-384" normalizes to SHA384. This unifies library calls (OpenSSL vs CryptoJS vs WebCrypto) under common primitives.

3. Cryptographic Entity Creation Identification: Detects generation of key pairs, nonces, message digests, and related cryptographic materials.

4. Normalize Detected Ciphers: Establishes unified naming independent of language (e.g., all cryptography.*.ec.ECDSA combinations and patterns map to ECDSA).

Detected ciphers, API calls, and entity information are stored in JSON format for downstream processing. Abstract crypto-related constructs into three node types: (1) CALL represents crypto API invocation (AES.new() , MessageDigest.getInstance("SHA1")), (2) ENT represents data entities (keys, IVs, nonces, constants), and (3) SYM represents symbolic identifiers (variables, parameters, aliases). This reduces language-specific syntax into a uniform crypto-specific IR JSON.

5.2. Step 2: Crypto Semantic Graph (CSG) and Backtracking

Now that we have three types of nodes, the next step is to add contextual information to these nodes. This happens in Step 2. In this step, the JSON graph is transformed into a Crypto Semantic Graph (CSG). A CSG is a semantic graph which stores rich entity information in nodes while edges represent relationships between nodes.

The CSG construction process is as follows:

1. Node type specification (CALL, ENT, SYM) with specific metadata is represented in the IR.

2. Edge creation: Shows connections between elements (e.g., API function calling specific library). Within a single function ("intra-procedural"), a def-use analysis on the IR graph is done to track where variables are defined and where they are used in function calls. Across functions ("inter-procedural"), we apply alias analysis to follow dependencies beyond function boundaries. We maintain mappings of variable aliases so that related operations remain connected even if variable names change. For example, if a key is generated (key = genKey()) and later passed to another call (signer.init(key)), these operations can be linked regardless of renaming.

3. Multihop backtracking to determine relationship directionality and trace cryptographic element origins: When a relationship can be established between the functions, an edge is added to the CSG. Otherwise, the edge can be temporarily marked as unresolved and can be optionally retried with deeper alias refinement. Edges can be constructed to capture the following semantic flows: (1) flows_to: data dependency (key passed to AES call), (2) produces: function output (AES.new → cipher object), (3) alias_of: variable alias tracking (key = masterKey), (4) taints: external/untrusted source flows (IV from user input). This forms a CSG of cryptographic interactions, not just isolated API calls to libraries.

Multihop backtracking traverses the Crypto Semantic Graph (CSG) backwards from a cryptographic sink (e.g., AES.encrypt, ECDSA.sign) to reconstruct the full provenance of all security-critical parameters (keys, nonces, salts, libraries, and related cryptographic materials). Each hop corresponds to a semantically meaningful dependency edge in the CSG.

```
        def backtrace_entities(graph, start_call, target_roles=("key","iv","nonce","salt"
)):
            Q = deque([start_call])
            visited = set()                          # check if node is already visited
            lineage = {"key": [], "iv": [], "nonce": [], "salt": []}
            while Q:
                node = Q.popleft()
                if node in visited:
                  continue
                  visited.add(node)
                for (src, dst, type) in graph.in_edges(node):
                    if type == "flows_to":             # follow dependencies (salt, key,
 iv, nonce)
                        ent = src
                        if ent.role in target_roles:
                          lineage[ent.role].append(ent)
                    for (p, e, t2) in graph.in_edges(ent):
                        if t2 == "produced_by":        # find who created this element
                            Q.append(p)
          return lineage
```

This finds all possible key/IV ancestors within a bounded hop count. It can detect, for example, that an IV entity with a constant origin comes from a hard-coded literal. In practice we limit depth (e.g. less than 50 hops) and stop at function/class boundaries.

5.3. Example Use Case Output

Consider a developer using cryptographic libraries to encrypt API traffic to web servers. The application must import cryptographic libraries, invoke algorithm-specific functions, and generate required keys. In the semantic graph JSON output, FLAIR detects the called algorithms. A post-processing step assembles file graphs into a repo-level CSG. Findings are emitted in JSON or SARIF format: each includes file, line, primitive, hash, OID, confidence, tags (e.g. STATIC_IV), and the key/IV lineage summary. Example finding:


```
        {
          "file": "src/crypto/sign.py",
          "line": 42,
          "primitive": "ECDSA",
          "hash": "SHA256",
          "risk_tags": [],
          "lineage": {
            "key": [{"origin":"gen_keypair","curve":"secp256r1"}],
            "iv": []
          },
          "explanation": "ECDSA(SHA256) with EC P-256 key; lineage consistent."
        }
```

The graph reveals source file, utilized algorithms, and associated key pairs. If the function used SHA1 instead of SHA256, FLAIR would detect this deprecated algorithm violation. The language-independence of this approach means cryptographic discovery is unaffected by function nomenclature or library switching, providing superior robustness to traditional pattern-matching approaches. The output can be stored in a database for further analysis or reporting. The output can also be extended to flag compliance violations based on organizational policies (e.g., deprecated algorithms, weak key sizes, etc.).

Organizations implementing FLAIR should will have to write custom rules for their cryptographic policies. Additional operational considerations include:

* Establishment of cryptographic standards and algorithm lifecycle policies

* Integration of FLAIR output with existing security scanning and policy enforcement pipelines

* Training for security and development teams on interpreting semantic graph outputs

* Periodic updates to detection rules for emerging cryptographic algorithms and threats

6. Security Considerations

The framework does not itself process sensitive cryptographic material; it analyzes code patterns and dependencies. Users should ensure appropriate access controls on source code repositories and generated reports containing cryptographic dependency information.

7. Conclusion

FLAIR represents a significant advancement in cryptographic element discovery by providing language-agnostic, semantic graph-based analysis that overcomes limitations of existing general-purpose and language-specific tools. By enabling comprehensive identification of cryptographic usage across polyglot enterprise environments, FLAIR facilitates compliance with cryptographic standards, detection of policy violations, and proactive migration to quantum-safe cryptography.

The framework’s targeted approach to graph optimization ensures scalability across large codebases while maintaining the precision necessary for accurate security analysis and policy enforcement.

8. References

[ref1] Xiao, Y., Zhao, Y., Allen, N., Keynes, N., Yao, D., Cifuentes, C., and ACM Digital Library, "Industrial Experience of Finding Cryptographic Vulnerabilities in Large-scale Codebases", 2023, <https://dl.acm.org/doi/full/10.1145/3507682>.

[ref2] Wickramasinghe, S. and Splunk, "Static Code Analysis: The Complete Guide to Getting Started with SCA", DOI 10.1145/3372802, 2025, <https://www.splunk.com/en_us/blog/learn/static-code-analysis.html>.

[ref3] Alshahrani, M., Khan, M., Maddouri, O., Kinjo, A., Queralt-Rosinach, N., and R. Hoehndorf, "Neuro-symbolic Representation Learning on Biological Knowledge Graphs", 2017, <https://pmc.ncbi.nlm.nih.gov/articles/PMC5860058/>.

Authors' Addresses

Golam Kayas
Comcast Corporation
Email: golam_kayas@comcast.com

Jagaputhran Sampathkumar
Comcast Corporation
Email: jagaputhran.s@gmail.com

Barath Sekar
Comcast Corporation
Email: barath_sekar@comcast.com

Sri Surya Gayatri B
Comcast Corporation
Email: srisuryagayatri_b@comcast.com

Jayati Dev
Comcast Corporation
Email: jayati_dev@comcast.com