# **A Comprehensive Framework for Normalizing and Deduplicating Security Findings**

## **1. Introduction**

### **The Challenge of Heterogeneous Security Findings**

Modern software development lifecycles (SDLCs) incorporate a diverse array of security assessment tools, including Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), Software Composition Analysis (SCA), Interactive Application Security Testing (IAST), infrastructure scanners, and more.1 While this multi-faceted approach enhances security coverage, it introduces a significant operational challenge: the heterogeneity of findings. Each tool often produces results in proprietary formats, employs distinct severity rating scales (e.g., Critical/High/Medium/Low, P1-P5, numerical scores), and describes vulnerabilities with varying levels of detail and structure.5

This lack of standardization creates substantial friction in vulnerability management workflows. Security teams expend considerable effort manually normalizing data, attempting to correlate findings from different sources, and struggling to gain a unified view of the organization's security posture.7 The inconsistency hinders effective prioritization, making it difficult to distinguish truly critical issues from noise. Furthermore, the high volume of raw findings, often including duplicates reported by multiple tools or multiple times by the same tool, contributes to alert fatigue and inefficient use of security resources.8 Without a systematic approach to normalization and deduplication, organizations risk overlooking critical vulnerabilities while drowning in redundant or low-priority data.

### **Objectives of the Unified Framework**

This report outlines a comprehensive framework designed to address the challenges posed by heterogeneous security findings. The primary objectives of this framework are:

1. **Normalization:** To ingest findings from diverse security assessment tools and transform them into a single, consistent, and unified format. This includes standardizing the data structure (schema) and severity ratings.
2. **Deduplication:** To accurately identify and consolidate findings that represent the same underlying vulnerability, regardless of the reporting tool or minor variations in the report details.
3. **Enrichment:** To augment normalized findings with crucial contextual information, such as exploitability data (e.g., CISA KEV, EPSS), asset criticality, code ownership, and historical data, enabling more accurate risk assessment.
4. **Prioritization:** To facilitate effective prioritization of remediation efforts based on a combination of normalized severity and contextual enrichment, moving beyond simplistic metrics like raw CVSS scores.6
5. **Efficiency:** To optimize the storage and retrieval mechanisms for handling potentially large volumes of security finding data, ensuring the framework remains performant and scalable.

By achieving these objectives, the framework aims to deliver significant value: improved accuracy in vulnerability assessment, drastic reduction in noise from duplicates and false positives, faster and more targeted remediation cycles, better allocation of security and development resources, and ultimately, an enhanced and more manageable application security posture.8

## **2. Core Framework Components**

This section details the essential components required to build a robust normalization and deduplication framework for security findings.

### **2.1. Common Schema Definition**

Rationale for a Unified Schema:

The cornerstone of managing findings from disparate sources is a unified schema. Attempting to process, correlate, and analyze findings directly in their native, tool-specific formats is inefficient and error-prone. A common schema provides a consistent structure, acting as a lingua franca for all security findings ingested into the framework.7 This standardization is fundamental for enabling effective aggregation, comparison, filtering, deduplication, enrichment, and reporting across the entire vulnerability dataset. It transforms a collection of isolated data points into a cohesive information system.

Analysis of Standard Formats:

Several industry standards exist for representing security findings or related information. Understanding their strengths and limitations is crucial when designing the internal common schema:

* **SARIF (Static Analysis Results Interchange Format):** An OASIS standard specifically designed for the output of static analysis tools.12 It offers a well-defined structure for reporting results (result objects), the rules that generated them (reportingDescriptor objects), precise code locations (physicalLocation), and mechanisms for result matching like fingerprints (fingerprints, partialFingerprints).12 Its adoption by tools 17 and platforms like GitHub Code Scanning 14 makes it a valuable input format. However, its primary focus is static analysis, and it might require adaptation to fully represent findings from DAST, SCA, or infrastructure scanners.
* **ASFF (AWS Security Finding Format):** The standard format used within AWS Security Hub to consolidate findings from various AWS services (like Macie, GuardDuty) and integrated third-party tools.19 ASFF provides a rich, detailed JSON structure encompassing finding types, severity, affected AWS resources (with detailed attributes), remediation information, and product-specific fields.19 Its strength lies in its deep integration with the AWS ecosystem, but its structure is inherently AWS-centric and may not map perfectly to non-AWS environments or all tool types.
* **CycloneDX VEX (Vulnerability Exploitability eXchange):** While not a general finding format, VEX is a specialized capability within the CycloneDX SBOM standard designed to communicate the exploitability status of a vulnerability within a specific product context.22 It includes fields for state (e.g., not\_affected, exploitable), justification, and response.22 Its concepts are valuable for the enrichment phase, but its structure is not intended for representing the initial finding itself.
* **Custom Schema:** Offers maximum flexibility to tailor the schema precisely to an organization's specific tools, workflows, and enrichment needs. The drawback is the lack of inherent interoperability compared to standard formats and the effort required for design and maintenance.

While these standards provide valuable structures and concepts, no single existing standard typically serves as a perfect, universal internal schema for *all* types of security findings and the desired contextual enrichment. SARIF is primarily SAST-oriented 12, ASFF is AWS-specific 19, and VEX focuses narrowly on exploitability status.22 A practical approach often involves defining a custom internal schema that draws inspiration from these standards, selecting the most relevant fields, and adding organization-specific attributes (e.g., business impact, application owner). This internal schema then acts as the target for normalization, with adapters built to map incoming data *from* various tool formats (including SARIF or ASFF where applicable) *into* this unified internal structure. This provides the optimal balance of leveraging standardization where possible while retaining the flexibility needed for comprehensive vulnerability management.

Proposed Core Schema Fields:

A robust common schema should include fields that capture essential information from the source tool, facilitate normalization and deduplication, and accommodate enrichment data. The following table outlines a proposed set of core fields:

**Table 1: Core Common Schema Fields**

| **Field Name** | **Data Type** | **Description** | **Example** | **Mapping Notes (Examples)** |
| --- | --- | --- | --- | --- |
| finding\_id | String (UUID) | Unique identifier generated by the framework for this finding instance. | f47ac10b-58cc-4372-a567-0e02b2c3d479 | Internal primary key. |
| original\_finding\_id | String | Identifier assigned by the source tool (if available). | TOOL-X-12345 | Helps trace back to the source report. |
| tool\_name | String | Name of the assessment tool that generated the finding. | Semgrep, Grype, Bandit | Essential for context and confidence scoring. |
| scanner\_confidence | Float (0.0-1.0) | Confidence score provided by the source tool (normalized to 0-1 scale if possible). | 0.85 | Input to framework's confidence score. Map from SARIF result.properties.confidence. |
| vulnerability\_id | String | Standardized vulnerability identifier (e.g., CVE, CWE, tool-specific rule ID). | CVE-2021-44228, CWE-79, eslint/no-eval | Normalize tool-specific IDs where possible. Maps to SARIF result.ruleId. |
| vulnerability\_name | String | Human-readable name or title of the vulnerability/rule. | Log4Shell, Cross-Site Scripting (XSS) | Maps to SARIF rule.name, ASFF Title. |
| description | String (Text) | Detailed description of the finding provided by the tool. | Remote code execution vulnerability... | Maps to SARIF result.message.text, ASFF Description. |
| severity\_original | String | Severity rating as reported by the source tool. | Critical, High, 7.8, P1 | Raw input for normalization. |
| severity\_normalized | String / Float | Normalized severity (e.g., 'Critical', 'High', 'Medium', 'Low', 'Info' or a CVSS score). | Critical, 9.8 | Result of severity normalization (Section 2.2). Maps to ASFF Severity.Label/Normalized. |
| status | String (Enum) | Current workflow status of the finding within the framework. | New, Triaged, False\_Positive, Mitigated | Managed by the framework/users. Maps to ASFF Workflow.Status. |
| confidence\_score | Float (0.0-1.0) | Calculated confidence score indicating likelihood of being a true positive (Section 2.4). | 0.95 | Calculated by the framework. Maps to ASFF Confidence. |
| location | Object | Structured data detailing where the finding was located (Section 2.3). | { "file\_path": "src/auth.py", "line": 75 } | Maps to SARIF result.locations, ASFF Resources. |
| fingerprint | String (Hash) | Unique, stable hash calculated for deduplication (Section 2.5). | a3f1b... | Calculated by the framework. Related to SARIF fingerprints. |
| is\_duplicate | Boolean | Flag indicating if this finding is considered a duplicate of another. | true | Set by the deduplication engine. |
| duplicate\_of\_finding\_id | String (UUID) | Reference to the finding\_id of the primary finding if is\_duplicate is true. | e5d3b11a-67dd-4a83-b678-1f13c3d4e580 | Link used by deduplication engine. |
| first\_seen\_timestamp | DateTime (ISO8601) | Timestamp when a finding with this fingerprint was first ingested. | 2023-10-26T10:00:00Z | Maps to ASFF CreatedAt. |
| last\_seen\_timestamp | DateTime (ISO8601) | Timestamp when this specific finding instance was last ingested (or seen in a scan). | 2023-10-27T14:30:00Z | Maps to ASFF UpdatedAt. |
| enrichment\_data | Object (JSON) | Container for additional contextual data gathered during enrichment (Section 2.8). | { "epss": 0.95, "cisa\_kev": true,... } | Populated by the enrichment process. |
| raw\_finding | Object (JSON) | Original, unmodified finding data as received from the source tool (optional). | {...tool specific json... } | Useful for debugging or accessing tool-specific details not in the common schema. |

Schema Extensibility:

The defined schema should serve as a core, but it must be extensible. Organizations will inevitably have specific needs, such as tracking application ownership, linking findings to specific sprints or releases, or adding custom risk scores. Using a flexible storage format like JSON (often within database types like JSONB in PostgreSQL 38 or native JSON in Elasticsearch 39) allows for easily adding custom fields or nested objects without requiring rigid schema migrations. The enrichment\_data object is a prime candidate for such custom extensions.

### **2.2. Severity Normalization**

Challenges with Diverse Rating Systems:

A significant hurdle in aggregating security findings is the inconsistent representation of severity across different tools.5 One tool might use qualitative labels like "Critical," "High," "Medium," "Low," while another uses a numerical scale (e.g., 1-10 or 0-100), and yet another might employ a priority system like P1-P5. This heterogeneity makes it impossible to perform meaningful cross-tool comparisons or consistent prioritization without a normalization step. Simply sorting by the original severity string yields chaotic and unreliable results.

Leveraging CVSS as a Baseline:

The Common Vulnerability Scoring System (CVSS) provides an industry-standard, open framework for assessing the characteristics and severity of vulnerabilities.5 CVSS (versions 3.1 and the newer 4.0) calculates a numerical score (0.0-10.0) based on a set of base metrics that quantify intrinsic characteristics:

* **Exploitability Metrics:** Attack Vector (AV), Attack Complexity (AC), Privileges Required (PR), User Interaction (UI).
* **Scope Metric (S):** Whether a vulnerability can impact resources beyond its own security scope.
* **Impact Metrics:** Confidentiality (C), Integrity (I), Availability (A) impact. 6

Using CVSS as a common denominator provides a standardized, quantitative baseline for severity. Many security tools already provide CVSS scores for CVE-identified vulnerabilities. However, CVSS has recognized limitations: the base score is static and does not account for temporal factors (e.g., exploit availability, patches) or environmental context (e.g., asset criticality, mitigating controls).6 Therefore, while CVSS is an excellent *baseline* for normalization, it should not be the sole factor in final prioritization.

Mapping Strategy and Best Practices:

An effective severity normalization strategy involves mapping all incoming findings to a consistent internal scale, using CVSS as the preferred reference point.

1. **Define Internal Scale:** Establish a clear, standardized internal severity scale. This could be qualitative (e.g., Critical, High, Medium, Low, Informational) or quantitative (e.g., mapping directly to CVSS 0-10). A qualitative scale is often more intuitive for reporting and triage.
2. **Develop Mapping Policy:**
   * **CVSS-Based Mapping:** If a tool provides a CVSS score (v3.1 or v4.0), map it directly to the internal scale based on defined ranges (e.g., 9.0-10.0 maps to Critical, 7.0-8.9 to High, etc.).5
   * **Tool-Specific Qualitative Mapping:** For tools providing qualitative ratings (e.g., 'High') but no CVSS score, create explicit mappings to the internal scale (e.g., Tool X 'High' -> Internal 'High'). This requires understanding the tool's specific definitions.
   * **Tool-Specific Quantitative Mapping:** For tools providing custom numerical scores, establish mapping rules to the internal scale or attempt to translate them to an approximate CVSS equivalent if possible.
   * **Handling Missing Severity:** Define a default severity (e.g., 'Medium' or 'Unknown') for findings where the source tool provides no severity information, flagging them for manual review.
3. **Prioritize CVSS:** When multiple severity indicators are present (e.g., a tool provides both 'High' and a CVSS score of 8.5), prioritize the CVSS score for consistency.
4. **Allow Overrides:** The normalized severity is a baseline. The framework must allow this baseline to be adjusted later based on contextual enrichment (see Section 2.8). A high CVSS score might be downgraded if exploitability is low (low EPSS) and the asset is non-critical, while a medium CVSS score might be upgraded if it's on a critical, internet-facing asset and listed in CISA KEV.6
5. **Documentation:** Clearly document all mapping rules and the internal severity scale definitions.

Common pitfalls include losing granularity by mapping everything to a very coarse scale too early, ignoring the context limitations of CVSS, and inconsistent mapping across different tools.6 The best practice is a two-stage process: first, normalize to a consistent baseline (preferably CVSS-aligned) 40; second, refine this baseline score during the enrichment and prioritization phases using contextual data. This approach preserves the richness of the original data while enabling context-aware risk assessment.

**Table 2: Example Severity Mapping Policy**

| **Tool Name** | **Tool Severity Input** | **CVSS v3.1 Score** | **Mapping Logic** | **Normalized Internal Severity** |
| --- | --- | --- | --- | --- |
| Grype | Critical | 9.5 | Use CVSS score. Map 9.0-10.0 to Critical. | Critical |
| Semgrep | ERROR | N/A | Map tool's 'ERROR' level to Internal 'High'. | High |
| Bandit | High | N/A | Map tool's 'High' level to Internal 'High'. | High |
| Tool X | Medium | 5.5 | Use CVSS score. Map 4.0-6.9 to Medium. | Medium |
| Tool Y | P3 | N/A | Map tool's 'P3' level to Internal 'Medium'. | Medium |
| Tool Z | N/A | N/A | No severity provided. Default to 'Medium' (flag). | Medium |
| Grype | Low | 3.1 | Use CVSS score. Map 0.1-3.9 to Low. | Low |
| Scanner A | Info | 0.0 | Map 0.0 or qualitative 'Info' to Informational. | Informational |

### **2.3. Location Mapping for Source Code Findings**

Standard Line/Column Mapping:

The most common method for specifying the location of a finding within source code involves identifying the file path and the specific line number(s) where the issue occurs. More precise locations also include start and end column numbers. Standard formats like SARIF explicitly support this through the physicalLocation object, which contains an artifactLocation (for the file URI) and a region (with startLine, startColumn, endLine, endColumn properties).12 This method is straightforward and easily understood by developers.

Challenges with Refactoring:

The primary drawback of line/column-based location mapping is its fragility. Simple code changes, such as adding or deleting lines, inserting comments, or reformatting code blocks, can shift the line numbers, causing the stored location information to become inaccurate. More significant refactoring, like moving functions between files or renaming variables, completely invalidates simple line/column tracking.46 This fragility hinders the ability to track vulnerabilities over time and is a major cause of failures in finding deduplication, as the same logical vulnerability might appear at different line numbers in subsequent scans.

Resilient Mapping using Abstract Syntax Trees (ASTs):

Abstract Syntax Trees (ASTs) offer a more robust approach to locating findings. An AST represents the hierarchical structure of source code, abstracting away superficial details like whitespace, comments, and exact line numbers.52 Nodes in the AST correspond to code constructs like functions, classes, statements, expressions, and variables.

By mapping a vulnerability to a specific node or path within the AST, its location becomes more resilient to code changes that don't alter the fundamental structure of the relevant code block.54 For example, a finding could be anchored to the AST node representing a specific function call or a particular variable declaration within a method. Techniques include:

* Hashing the content or structure of the relevant AST node.
* Using a path descriptor from the root of the AST to the target node.
* Considering the context of the node (parent nodes, sibling nodes) in the fingerprint. 54

Other Resilient Techniques:

Beyond ASTs, other techniques can contribute to resilient location tracking:

* **Code Similarity Hashing:** Applying fuzzy hashing algorithms (like ssdeep) or other code similarity techniques (e.g., based on token sequences or n-grams 57) to the code snippet associated with the finding. This can help identify the same or very similar code blocks even if they have moved or undergone minor modifications.
* **Logical Signatures:** Defining signatures based on data flow or control flow graphs derived from the code.58 A vulnerability might be identified by a specific pattern of data originating from a source and flowing to a sink, which can be more stable than exact code patterns or line numbers.

Mapping Accuracy Considerations:

Achieving perfect location tracking through significant code refactoring remains a complex challenge.48 While ASTs and code hashing provide significantly more resilience than simple line numbers, they are not foolproof. Generating and parsing ASTs consistently across different languages, language versions, and parsing tools can be difficult. Furthermore, major architectural changes can alter AST structures significantly. The trade-off is between the simplicity and directness of line numbers and the complexity but greater resilience of ASTs or code hashing.

Proposed Approach:

The framework should store the standard line/column information provided by the tool (location.file\_path, location.line, location.column) as this is essential for displaying the finding to developers in their IDEs or code viewers. However, for the purpose of robust tracking and deduplication, the finding fingerprint (Section 2.5) should incorporate elements derived from more resilient techniques like AST node identifiers/hashes or code context hashes.

The primary value derived from AST analysis or code hashing in this context is not necessarily to update the displayed line number perfectly after refactoring, but rather to ensure that the framework can reliably recognize that a finding at file.js:20 in the current scan is the *same logical finding* as the one previously reported at file.js:15. This logical identification is crucial for accurate deduplication. The framework should store the *most recently reported* line/column information for display purposes, while relying on the stable fingerprint for maintaining the finding's identity across scans.

### **2.4. Confidence Scoring Methodology**

Defining Confidence:

A confidence score quantifies the likelihood that a finding reported by a security tool represents a genuine, exploitable security issue, as opposed to a false positive.60 It is distinct from severity, which measures the potential impact of the vulnerability if exploited. A high-severity finding might have low confidence if the detection method is prone to errors, while a low-severity finding could have high confidence if its existence is definitively confirmed. Confidence scoring helps security teams prioritize triage efforts, focusing first on findings that are most likely to be real.60

Sources of Uncertainty:

Confidence levels vary due to several factors inherent in security scanning:

* **Tool Limitations:** Different tools employ different analysis techniques (e.g., simple pattern matching vs. deep data-flow analysis vs. runtime probing). Some techniques are inherently more precise than others.3
* **Rule Precision:** The specific rule or signature used to detect a finding can vary in its accuracy. Broad or heuristic rules are more likely to generate false positives than highly specific ones.62
* **Analysis Context:** Static analysis (SAST) often lacks runtime context, potentially flagging issues in unreachable code, while dynamic analysis (DAST) might miss vulnerabilities in code paths not exercised during the test.3
* **Environmental Factors:** Tools typically analyze code or applications in isolation, without full knowledge of mitigating controls, specific configurations, or the runtime environment, which can affect actual exploitability.61

Factors Influencing Confidence Score:

A robust confidence score should synthesize information from multiple sources:

* **Rule Precision/Type:** Assign higher baseline confidence to findings from rules known to be precise (e.g., those based on taint analysis identifying source-to-sink flows) compared to less precise rules (e.g., simple regex matches for potentially sensitive patterns).62
* **Tool Reputation/Type:** Factor in the known reliability and methodology of the source tool. Findings from DAST or IAST tools, which often test running applications, might warrant higher initial confidence for certain vulnerability classes compared to SAST.62 Tools with a strong track record might receive a higher baseline confidence.
* **Historical Accuracy (Feedback Loop):** Crucially, incorporate data from past manual triage efforts (Section 2.7). If a specific rule or tool consistently generates findings marked as false positives by analysts, its confidence score should be dynamically lowered over time. Conversely, rules with a high true positive rate should have their confidence boosted.62
* **Evidence Clarity:** Evaluate the quality of evidence provided by the tool. A finding with a clear, verifiable exploit trace or data flow path should have higher confidence than one with vague or circumstantial evidence.62
* **Sensor/Agent Confirmation:** Give significantly higher confidence to findings confirmed by runtime instrumentation, such as IAST agents or SAST tools enhanced with runtime sensors (like Acunetix's AcuSensor 61). These tools can often confirm reachability and data flow in the live application.

**Proposed Scoring Model:**

1. **Scale:** Define a quantitative scale, such as 0.0 to 1.0 (or 0-100), where higher values indicate greater confidence. Alternatively, use qualitative levels (e.g., High, Medium, Low Confidence). A quantitative scale allows for finer granularity.
2. **Baseline Score:** Assign a baseline confidence score to each rule or finding type based on the tool and rule precision/type. For example, a taint-analysis rule from a reputable SAST tool might start at 0.8, while a simple pattern match might start at 0.5. Findings confirmed by DAST/IAST might start higher, e.g., 0.9 or 0.95.61
3. **Adjustment Factors:** Develop multipliers or additive factors based on other available information:
   * *Historical Accuracy:* Adjust the baseline score based on the historical True Positive Rate (TPR) for that specific rule/tool combination derived from the feedback loop. Confidenceadjusted​=Confidencebaseline​×TPRhistorical​.
   * *Evidence Quality:* Apply a small boost for findings with clear evidence (e.g., +0.05) or a penalty for vague evidence (-0.05).
   * *Sensor Confirmation:* Apply a significant boost if confirmed by a runtime sensor (e.g., set confidence to 0.99).
4. **Final Score:** Combine the baseline and adjustments to produce the final confidence score.

The true power of confidence scoring lies in its dynamic nature. Initial estimates based on tool type or rule complexity provide a starting point, but these are inherently static.67 The feedback loop described in Section 2.7 is essential. By systematically tracking manual triage results (True Positive, False Positive) against the specific rule ID, tool, and potentially even the code context (fingerprint), the system can learn over time. This allows for the statistical adjustment of future confidence scores, making the system progressively better at distinguishing likely true positives from noise, thereby significantly improving the efficiency of the triage process.69

### **2.5. Finding Fingerprinting for Deduplication**

Fingerprinting Principles:

The core objective of fingerprinting in this context is to generate a unique and stable identifier for each distinct security vulnerability instance.57 This identifier, or "fingerprint," should remain consistent even if the finding is reported by different tools, at different times, or if minor, non-functional code changes occur in the vicinity. It serves as the primary key for deduplication.57 Effective fingerprinting relies on hashing algorithms to condense multiple identifying characteristics into a single, fixed-size string.57

Key Fingerprint Components:

A robust fingerprint must incorporate multiple facets of the finding to ensure both uniqueness and stability:

1. **Vulnerability Type (Normalized):** This is the most critical component. It should represent the specific nature of the flaw.
   * Use a standardized identifier like a CWE ID (e.g., CWE-79) where applicable.82
   * For tool-specific rules without a standard mapping, use the tool's precise rule ID (e.g., semgrep-rule-xss-123).
   * Avoid relying solely on CVE IDs, as one CVE can manifest in multiple ways or be reported differently by various tools. Use the most specific identifier available.
2. **Location (Normalized):** Pinpointing where the vulnerability exists.
   * **File Path:** Use a normalized path, typically relative to the repository root, to ensure consistency across different checkout locations. Normalize path separators (e.g., always use /).
   * **Code Construct Identifier:** Simple line numbers are too fragile.46 Use more stable identifiers:
     + Function/Method/Class Name: The name of the enclosing function or class.
     + AST Node Path/Hash: A representation of the specific AST node or its path from the root, resistant to line number changes.54
3. **Code Context/Snippet:** Capturing the essence of the vulnerable code itself.
   * **Source Code Hash:** A hash (e.g., SHA-256) of the specific line(s) of code identified by the scanner. Normalize the code first by removing whitespace and comments to improve stability against formatting changes.82
   * **AST Subtree Hash:** A hash of the relevant AST subtree provides even greater resilience to minor code modifications that preserve the structure.82

Algorithms and Robustness:

Standard cryptographic hash functions like SHA-256 are suitable for generating the final fingerprint from the concatenated components.80 The process involves:

1. Extracting the relevant components (Normalized Type, Normalized Location, Context Hash).
2. Concatenating these components into a canonical string format.
3. Hashing the resulting string using SHA-256 (or a similar secure hash).

For code context, techniques like context-aware hashing (ignoring whitespace, comments, potentially variable names) or even perceptual hashing (if needing to match syntactically different but semantically similar code snippets) can enhance robustness, though they add complexity.57

Leveraging SARIF Fingerprints:

The SARIF standard includes properties specifically designed for result identification: fingerprints and partialFingerprints.13

* fingerprints: Intended as a stable identifier, often computed by the result management system.16
* partialFingerprints: Allows the *producing tool* to provide components that contribute to the fingerprint (e.g., a hash of the code pattern matched, context information). Keys are versioned strings (e.g., mytool/v1/codeHash). 16

When a tool provides partialFingerprints, the framework should leverage this information. It can combine these partial fingerprints (potentially with other components like normalized location) to compute the final internal fingerprint, respecting the versioning indicated by the keys.16

A single, monolithic fingerprint derived from concatenating all components can be brittle. If any single component changes (e.g., slight code refactoring changes the context hash, or a tool update changes a rule ID), the entire fingerprint changes, breaking deduplication. A more resilient strategy involves computing and potentially storing hashes for individual components or logical groups of components (Type+Location, CodeContext). This aligns well with the concept of SARIF partialFingerprints. The deduplication engine (Section 2.6) can then use more sophisticated logic, matching findings if a sufficient subset of these partial fingerprints align, even if one component differs. This composite approach provides greater robustness against minor variations compared to relying on a single hash value.13

**Table 3: Fingerprint Component Examples**

| **Component Type** | **Description** | **Example Value** | **Normalization Notes** |
| --- | --- | --- | --- |
| Vulnerability Type | Most specific normalized identifier (CWE/Rule ID) | CWE-79 | Use CWE if available, else precise tool rule ID. |
| Location Path | File path relative to repository root | src/app/controllers/userController.js | Normalize separators to /; ensure consistent root. |
| Location Anchor | Function/Method name or AST Node Identifier | processUserData, AST:/path/to/node | Extract from code/AST; provides stability over line #. |
| Code Context Hash | SHA-256 hash of normalized code snippet/AST node | a3f1bcd... | Normalize code (trim whitespace, ignore comments) first. |

### **2.6. Deduplication Engine**

Objective:

The core function of the deduplication engine is to accurately identify and link multiple finding reports that refer to the same underlying security issue.8 This prevents the inflation of vulnerability counts and ensures that remediation efforts are focused on unique problems. Effective deduplication must handle findings from different tools, repeated findings from the same tool across scans, and findings that persist despite minor code changes.

Rule-Based Deduplication:

This approach relies on predefined logic to determine if two findings are duplicates.

1. **Exact Fingerprint Matching:** This is the primary and most reliable method. If two findings possess identical fingerprints (as generated in Section 2.5), they are considered duplicates.9 The finding ingested first is typically marked as the "original," and subsequent matches are marked as duplicates linked to the original.
2. **Context-Aware Rules:** More sophisticated rules can consider the context surrounding a finding. For instance:
   * *Proximity Rule:* If two findings of the same type (e.g., CWE-79) occur within the same function or code block (information potentially derived from AST analysis or code structure parsing) but have slightly different line numbers or code snippets (leading to different fingerprints), a rule might flag them as *potential* duplicates requiring review, or even automatically link them if confidence is high.95
   * *Asset Context Rule:* If the same CVE is reported by an infrastructure scanner on multiple virtual machines that are known clones or part of the same auto-scaling group, a rule might consolidate these into a single logical issue linked to the group/template, rather than treating each as entirely separate. This requires integration with asset inventory/CMDB data.
3. **Superset/Subset Logic:** This addresses scenarios where one finding is a more specific instance of another reported finding. For example, a SAST tool might report a general "Tainted data flow to SQL execution sink" (e.g., CWE-89) and also a more specific rule match "SQL injection via parameter X in function Y." Handling this requires semantic understanding of the rules. A rule-based system might define relationships (e.g., "Rule Z is a specification of CWE-89") to link these findings or prioritize the more specific one.95 This is complex and often requires manual curation of rule relationships or leveraging ML techniques.
4. **Rule ID Mapping:** Before comparing fingerprints, rules might be needed to map equivalent vulnerability identifiers from different tools to a common ID (e.g., map Tool A's rule\_123 and Tool B's xss\_check both to CWE-79).

Machine Learning Approaches:

ML techniques can overcome the rigidity of rule-based systems, particularly when dealing with variations in descriptions, code snippets, or locations that cause fingerprints to differ slightly.11

1. **Natural Language Processing (NLP) for Semantic Similarity:**
   * *Vectorization:* Convert textual elements of findings (e.g., description, vulnerability\_name, relevant code snippets) into numerical vectors using techniques like TF-IDF, word embeddings (Word2Vec, GloVe), or more advanced transformer models (BERT, domain-specific models like Foundation-Sec 102).103 These vectors capture the semantic meaning of the text.
   * *Similarity Calculation:* Compute the similarity between these vectors using metrics like cosine similarity.104 Findings with vector similarity above a predefined threshold are considered potential duplicates.103
2. **Clustering:**
   * Apply clustering algorithms (e.g., K-Means, DBSCAN, hierarchical clustering 113) to the generated finding vectors.103 Findings grouped within the same cluster are likely related and represent potential duplicates. This can uncover groups of similar issues that might be missed by pairwise similarity checks.

Hybrid Approach (Recommended):

A combination of rule-based and ML approaches generally yields the best results:

1. **Phase 1 (High Confidence):** Apply exact fingerprint matching. Findings with identical fingerprints are marked as duplicates with high confidence.
2. **Phase 2 (Potential Duplicates):** For findings not matched in Phase 1, apply ML-based semantic similarity (NLP vectorization + similarity threshold/clustering) to identify findings that are textually or contextually similar despite differing fingerprints.
3. **Handling ML Matches:** Findings identified as duplicates solely through ML methods should be treated with caution due to the probabilistic nature of these techniques.103 Instead of automatically merging them, they could be:
   * Flagged as "potential duplicates" requiring manual review and confirmation.
   * Linked with a lower confidence score or a specific "ML-identified duplicate" status.
   * Used as a secondary signal to potentially link findings if some, but not all, parts of a composite fingerprint match.

This hybrid strategy leverages the certainty of exact fingerprint matching for clear duplicates while using the flexibility of ML to catch semantic similarities, acknowledging and managing the inherent uncertainty of ML predictions.

Managing Duplicate Sets:

The framework needs a clear strategy for managing duplicates. A common approach, similar to DefectDojo 9, is to designate one finding instance as the "primary" or "original" and link all subsequent duplicate findings to it. Triage actions (status changes, comments, assignments) are typically performed on the primary finding, implicitly applying to all linked duplicates. The is\_duplicate and duplicate\_of\_finding\_id fields in the common schema support this model.

### **2.7. False Positive Reduction**

The Cost of False Positives (FPs):

False positives – findings reported by tools that do not represent actual vulnerabilities – are a significant drain on security resources. They lead to alert fatigue, where analysts become desensitized and may overlook genuine threats.10 Investigating FPs consumes valuable time that could be spent on remediating true positives or proactive security measures. Furthermore, a high FP rate erodes developers' trust in security tooling, potentially hindering adoption and collaboration.10

**Techniques for FP Reduction:**

1. **Rule Tuning/Refinement:** Directly modifying the detection logic within security tools or the framework's custom rules. This might involve making regex patterns more specific, adjusting thresholds in heuristics, or adding exclusion criteria to rules. This requires deep understanding of the specific tool and rule logic.
2. **Contextual Analysis:** Automatically suppressing findings based on their context. Examples include:
   * *File Path Exclusions:* Ignoring findings in test directories, documentation folders, or third-party library code that is not directly modifiable.116
   * *Code Reachability:* Suppressing findings in code identified as unreachable through static analysis (if available).116
   * *Asset Status:* Ignoring findings on assets marked as decommissioned or belonging to non-production environments (requires CMDB integration).10
3. **Machine Learning for FP Prediction:** Training classification models to predict the likelihood that a finding is a false positive based on its features (e.g., rule ID, code patterns, location, historical data).10 Findings predicted as FPs with high confidence can be automatically suppressed or deprioritized.
4. **Allow-listing/Manual Suppression:** Providing a mechanism for analysts to manually mark specific, verified false positives.116 These suppressions should be tracked based on the finding's fingerprint to automatically ignore recurrences of the *exact same* FP in future scans.

Implementing Feedback Loops:

A crucial component of sustainable FP reduction is establishing a feedback loop where manual triage decisions inform and improve the automated system.69

1. **Capture Triage Data:** The vulnerability management interface (UI or API) must allow analysts to explicitly mark findings as "True Positive," "False Positive," or potentially other states like "Risk Accepted."
2. **Record Justification:** When marking an FP, it is vital to capture the *reason* for the classification. Common reasons might include: "Test Code," "Code Not Reachable," "Mitigating Control Exists," "Inaccurate Detection Logic," "Acceptable Business Risk."
3. **Feed Data Back:** This structured feedback data should be periodically or continuously fed back into the system to:
   * **Refine Confidence Scores (Section 2.4):** Automatically lower the calculated confidence score for rules or tools that consistently produce findings marked as FPs for specific reasons.
   * **Improve Deduplication Models:** Use confirmed true/false positive labels to retrain ML models used for semantic similarity, helping them better distinguish between truly distinct issues and mere variations.
   * **Suggest Rule Tuning:** Identify rules with high FP rates and flag them for manual review and potential tuning by security engineers. The justification provided helps guide the tuning process.
   * **Maintain Suppression Lists:** Automatically add the fingerprints of confirmed FPs to a suppression list, preventing them from appearing in future reports unless the underlying code changes significantly (invalidating the fingerprint).

Capturing the *reason* for marking a finding as a false positive provides much richer information than a simple binary label. Knowing *why* a rule misfired allows for more targeted and effective improvements to the automated detection and scoring mechanisms, leading to a more efficient and trusted vulnerability management process over time.116

### **2.8. Finding Enrichment**

Importance of Context:

Raw findings generated by security tools often lack the necessary context for accurate risk assessment and effective prioritization.121 A reported vulnerability's technical severity (like a CVSS score) is only one piece of the puzzle. Understanding its real-world exploitability, the business criticality of the affected asset, and who is responsible for the code are crucial for determining the actual risk and urgency. Enrichment is the process of adding this vital contextual data to the normalized findings.

Integrating External Threat Intelligence:

Leveraging external data sources provides insights into the current threat landscape and the likelihood of a vulnerability being exploited.

* **CISA KEV (Known Exploited Vulnerabilities) Catalog:** This catalog, maintained by CISA, lists vulnerabilities that are known to be actively exploited in the wild.43 Checking if a finding's CVE is present in the KEV catalog is a critical enrichment step. A match significantly elevates the priority, regardless of the base CVSS score. Integration involves periodically downloading the KEV catalog (available as JSON/CSV) and checking for CVE matches.
* **EPSS (Exploit Prediction Scoring System):** EPSS provides a probabilistic score (0-100%) indicating the likelihood of a CVE being exploited in the next 30 days.44 This score is more dynamic than CVSS and considers factors related to exploit availability and attacker interest. Integration can be done programmatically via the EPSS API, querying by CVE ID to retrieve the current probability and percentile scores.125
* **Other Threat Intelligence (TI) Feeds:** Integrating commercial or open-source TI feeds can provide additional context, such as associated threat actors, malware families, attack campaigns, or indicators of compromise (IoCs) related to a vulnerability.122 This requires subscribing to feeds and correlating them with findings based on CVE or other identifiers.

Integrating Internal Context:

Internal organizational data provides context about the specific environment where the vulnerability exists.

* **Code Ownership (CODEOWNERS):** Identifying the team or individuals responsible for the affected code is crucial for efficient remediation assignment. Parsing CODEOWNERS files (common in platforms like GitHub and GitLab) based on the finding's file path allows automatic assignment or notification.126 Integration requires accessing the repository and parsing the relevant CODEOWNERS file according to its syntax.
* **Code Churn/Authorship (git blame):** Understanding who last modified the vulnerable code and how recently can provide context for investigation. Programmatically executing git blame for the specific file and line range associated with the finding can retrieve author information and commit details.132 Using the --porcelain or --line-porcelain output formats facilitates parsing.132 High churn in a file might indicate increased risk or recent introduction of the flaw.
* **Asset Inventory / CMDB Data:** Linking findings to specific assets (servers, containers, applications, databases) registered in a Configuration Management Database (CMDB) or asset inventory system is vital for assessing business impact.133 Information retrieved can include:
  + Asset Criticality (e.g., Tier 1, Production, Critical)
  + Environment (e.g., Production, Staging, Development)
  + Asset Owner / Responsible Team
  + Deployed Application/Service
  + Data Sensitivity Classification Integration typically involves querying the CMDB API using asset identifiers (like hostname, IP address, or application name) potentially derived from the scan target information.
* **Reachability Analysis:** Incorporating data about network exposure is critical. Is the vulnerable port/service accessible from the internet? Is it firewalled? This information might come from network scan results, infrastructure-as-code analysis, or CMDB data indicating network zone placement.116

Implementation Strategy:

The enrichment process should ideally run after a finding has been normalized and potentially deduplicated (at least based on initial fingerprinting). The gathered contextual information should be stored in the dedicated enrichment\_data field within the common schema object. The prioritization engine should then heavily weigh these enrichment factors alongside the normalized severity when determining the final risk score and remediation urgency.

Enrichment is the key step that elevates vulnerability management from a simple severity-based checklist exercise to a truly risk-based decision-making process. Combining external threat intelligence (likelihood of exploit) with internal business context (potential impact) provides a far more accurate picture of the actual risk posed by each finding.121 This allows organizations to focus remediation efforts on the vulnerabilities that present the greatest danger to their specific environment and business objectives, rather than chasing high CVSS scores that may have low real-world exploitability or impact.6

### **2.9. Data Storage and Retrieval Optimization**

Requirements:

A central repository for normalized and enriched security findings needs to meet several key requirements:

* **Scalability:** Capable of storing and managing potentially millions or even billions of finding records over time, originating from frequent scans across numerous assets.
* **Query Performance:** Support fast retrieval and aggregation for various use cases, including interactive dashboards (displaying trends, counts by severity/status), analyst triage queues (filtering by tool, status, assignee), and correlation analysis (finding related vulnerabilities across assets or time).
* **Flexibility:** Accommodate the common schema, including potentially nested enrichment data, and allow for schema evolution as new tools or enrichment sources are added.
* **Data Security:** Ensure the confidentiality, integrity, and availability of sensitive vulnerability data through appropriate access controls, encryption, and backup strategies.135

Database Options Analysis:

Choosing the right database technology is crucial for meeting these requirements. The main contenders include:

* **PostgreSQL (Relational Database):**
  + *Pros:* Mature, widely adopted, ACID compliant, strong support for relational data and joins, excellent support for structured data, and powerful JSONB capabilities for semi-structured data flexibility.39 Supports various indexing strategies for optimization.38
  + *Cons:* Can be less performant for unstructured full-text search compared to specialized search engines. Complex relationship queries involving many joins can become slow on very large datasets.137 Requires careful schema design and indexing for optimal performance.140
* **Elasticsearch (Search Engine / Document Store):**
  + *Pros:* Optimized for full-text search, log analytics, and aggregations; horizontally scalable; schema-flexible (schemaless or explicit mapping); excellent for building dashboards and performing fast searches across large volumes of text-heavy data.39 RESTful API is easy to integrate.142
  + *Cons:* Not designed for transactional integrity (less critical for findings data). Performing complex joins or graph-like traversals is difficult and often requires data denormalization, which can increase storage.145 Managing large clusters requires operational expertise.
* **Graph Databases (e.g., Neo4j):**
  + *Pros:* Natively designed to store and query relationships between entities. Excels at traversing complex connections (e.g., vulnerability -> exploits -> library -> used in -> application -> hosted on -> server -> exposed to -> internet).137 Ideal for attack path analysis, root cause analysis, and understanding complex dependencies.148
  + *Cons:* May be overkill if the primary need is simple filtering and aggregation. Can have a steeper learning curve for data modeling (nodes and edges) and querying (e.g., Cypher).137 May not be as optimized for full-text search or simple list retrieval as Elasticsearch or PostgreSQL.

**Schema Design and Indexing Strategies:**

Regardless of the database chosen, effective schema design and indexing are critical for performance.

* **Schema Design:**
  + *PostgreSQL:* Use appropriate data types (TEXT, VARCHAR, INTEGER, TIMESTAMP, JSONB for enrichment\_data and raw\_finding). Normalize related entities (like assets, rules) into separate tables if complex relationships need querying, but consider performance impact of joins. Leverage JSONB for flexibility within the main finding table.140
  + *Elasticsearch:* Define explicit mappings for key fields using types like keyword (for exact matching, aggregation, sorting on IDs, status, severity, tool name, fingerprint), text (for searching descriptions), integer/float (for line numbers, scores), date (for timestamps), and object/nested (carefully, for structured location/enrichment data).145 Denormalization (embedding related asset/rule info) is often preferred over joins to optimize search speed.145
  + *GraphDB (Neo4j):* Model findings, vulnerabilities (CVE/CWE), tools, files, code constructs, assets, and teams as nodes. Use relationships (e.g., :DETECTED\_BY, :LOCATED\_IN, :AFFECTS, :OWNED\_BY) to connect them. Store properties (severity, status, fingerprint) on the finding nodes or relationships.
* **Indexing:**
  + *General:* Index all fields frequently used in WHERE clauses, JOIN conditions, ORDER BY, and GROUP BY operations.
  + *PostgreSQL:* Use B-tree indexes for most fields (finding\_id, tool\_name, severity\_normalized, status, cve, file\_path, fingerprint). Use multicolumn indexes for common filter combinations (e.g., (tool\_name, status)).38 Consider GIN indexes for full-text search on descriptions or querying within JSONB fields.38
  + *Elasticsearch:* Fields mapped as keyword are automatically indexed efficiently for filtering and aggregation. text fields require appropriate analyzers for searching. Ensure numeric and date fields are mapped correctly for range queries. Optimize shard count and routing for large datasets.145
  + *GraphDB (Neo4j):* Create indexes on node properties frequently used as starting points for queries or in WHERE clauses (e.g., index on :Finding(fingerprint), :CVE(id), :Asset(name)).

Secure Storage Practices:

Protecting the vulnerability data itself is paramount. Implement:

* **Access Control:** Role-based access control (RBAC) at the database level and application level to restrict who can view or modify findings.136
* **Encryption:** Encrypt data both at rest (using database-level or filesystem-level encryption like AES) and in transit (using TLS/SSL).136
* **Backups:** Implement regular, automated backups following the 3-2-1 rule (three copies, two different media types, one offsite).135
* **Auditing:** Log access and modifications to finding data for accountability and forensic analysis. Consider immutable storage for audit logs if possible.136

No single database excels at *all* potential query patterns for security findings. Elasticsearch is often favored for its search and dashboarding capabilities, making it suitable for SIEM-like use cases. PostgreSQL offers robustness, relational integrity, and strong JSON support, making it a solid general-purpose choice. Graph databases provide unique advantages for analyzing complex relationships and attack paths. The best choice depends on the primary use cases. A hybrid approach (e.g., Elasticsearch for fast search/aggregation, PostgreSQL for relational storage and reporting) might be optimal but introduces complexity. Regardless of the chosen database(s), a well-designed schema and, critically, effective indexing strategies tailored to the expected query workload are essential for achieving performance at scale.38

## **3. Example Implementation (Python Pseudocode)**

This section provides high-level Python pseudocode examples to illustrate the core logic of the framework components. These examples focus on the conceptual flow and data transformations rather than specific library implementations.

Python

import hashlib  
import json  
import re  
from datetime import datetime  
  
# --- Placeholder for external data sources ---  
CISA\_KEV\_CATALOG = set(["CVE-2021-44228", "CVE-2023-1234"]) # Example  
EPSS\_API\_ENDPOINT = "https://api.first.org/data/v1/epss?cve={cve}"  
CMDB\_API\_ENDPOINT = "http://cmdb.example.com/api/assets?name={name}"  
CODEOWNERS\_PATH\_TEMPLATE = "/path/to/repo/{repo\_name}/CODEOWNERS"  
GIT\_REPO\_PATH\_TEMPLATE = "/path/to/repo/{repo\_name}"  
  
# --- 1. Common Schema Definition (Conceptual Class) ---  
class NormalizedFinding:  
 def \_\_init\_\_(self):  
 self.finding\_id = None # UUID generated later  
 self.original\_finding\_id = None  
 self.tool\_name = None  
 self.scanner\_confidence = None  
 self.vulnerability\_id = None # Normalized (e.g., CWE-XXX, CVE-YYYY-ZZZZ)  
 self.vulnerability\_name = None  
 self.description = None  
 self.severity\_original = None  
 self.severity\_normalized = None # E.g., 'Critical', 'High', or CVSS score  
 self.status = "New" # Default status  
 self.confidence\_score = None # Calculated later (0.0-1.0)  
 self.location = {  
 "file\_path": None, # Normalized path  
 "line": None,  
 "column": None,  
 # Potentially add AST node info, function name etc.  
 }  
 self.fingerprint = None # Calculated later  
 self.is\_duplicate = False  
 self.duplicate\_of\_finding\_id = None  
 self.first\_seen\_timestamp = None # Set on first ingestion of this fingerprint  
 self.last\_seen\_timestamp = None # Updated on every ingestion  
 self.enrichment\_data = {}  
 self.raw\_finding = None # Original tool output  
  
# --- 2. Normalization Functions ---  
  
def normalize\_sarif\_finding(sarif\_result, tool\_name):  
 """Maps a SARIF result object to the NormalizedFinding schema."""  
 finding = NormalizedFinding()  
 finding.tool\_name = tool\_name  
 finding.raw\_finding = sarif\_result # Store original  
  
 finding.vulnerability\_id = sarif\_result.get("ruleId", "UNKNOWN\_RULE")  
 if sarif\_result.get("message"):  
 finding.description = sarif\_result["message"].get("text", "")  
 # Potentially extract name from message if not explicit  
 finding.vulnerability\_name = finding.description.split('.') # Simple example  
  
 finding.severity\_original = sarif\_result.get("level", "unknown") # e.g., 'error', 'warning'  
  
 # Location mapping (simplified example, assumes one location)  
 if sarif\_result.get("locations"):  
 loc = sarif\_result["locations"].get("physicalLocation")  
 if loc and loc.get("artifactLocation"):  
 finding.location["file\_path"] = normalize\_path(loc["artifactLocation"].get("uri"))  
 if loc and loc.get("region"):  
 finding.location["line"] = loc["region"].get("startLine")  
 finding.location["column"] = loc["region"].get("startColumn")  
  
 # Extract scanner confidence if available (example property)  
 finding.scanner\_confidence = sarif\_result.get("properties", {}).get("confidence", None)  
  
 # Timestamps set during ingestion  
 finding.last\_seen\_timestamp = datetime.utcnow().isoformat() + "Z"  
  
 return finding  
  
def normalize\_grype\_finding(grype\_match, tool\_name):  
 """Maps a Grype match object to the NormalizedFinding schema."""  
 finding = NormalizedFinding()  
 finding.tool\_name = tool\_name  
 finding.raw\_finding = grype\_match  
  
 finding.vulnerability\_id = grype\_match["vulnerability"]["id"] # Usually CVE  
 finding.vulnerability\_name = f"{grype\_match['artifact']['name']}@{grype\_match['artifact']['version']} - {finding.vulnerability\_id}"  
 finding.description = grype\_match["vulnerability"].get("description", "")  
 finding.severity\_original = grype\_match["vulnerability"]["severity"] # e.g., 'Critical'  
  
 # Location (Package focused)  
 finding.location["file\_path"] = grype\_match["artifact"]["locations"]["path"] if grype\_match["artifact"].get("locations") else None # Path to manifest/package file  
 finding.location["package\_name"] = grype\_match["artifact"]["name"]  
 finding.location["package\_version"] = grype\_match["artifact"]["version"]  
 finding.location["package\_type"] = grype\_match["artifact"]["type"]  
  
 # Grype doesn't typically provide line/column or scanner confidence  
 finding.scanner\_confidence = None  
  
 # Timestamps set during ingestion  
 finding.last\_seen\_timestamp = datetime.utcnow().isoformat() + "Z"  
  
 return finding  
  
# Add similar normalization functions for other tools (Bandit, Semgrep, etc.)  
  
def normalize\_path(raw\_path, repo\_root="/path/to/repo/"):  
 """Normalizes a file path to be relative to a conceptual repo root."""  
 if not raw\_path:  
 return None  
 # Basic normalization: remove scheme, make relative, use forward slashes  
 path = re.sub(r'^file://', '', raw\_path)  
 if path.startswith(repo\_root):  
 path = path[len(repo\_root):]  
 return path.replace('\\', '/')  
  
def normalize\_severity(original\_severity, tool\_name, cvss\_score=None):  
 """Maps original severity to an internal scale using CVSS as baseline."""  
 # Simplified example mapping  
 internal\_scale = {"Critical": 9.5, "High": 8.0, "Medium": 5.5, "Low": 2.0, "Info": 0.0, "Unknown": 5.0} # Map internal names to approx CVSS midpoint  
 qualitative\_internal\_scale = {  
 (9.0, 10.0): "Critical",  
 (7.0, 8.9): "High",  
 (4.0, 6.9): "Medium",  
 (0.1, 3.9): "Low",  
 (0.0, 0.0): "Informational"  
 }  
  
 if cvss\_score is not None:  
 try:  
 score = float(cvss\_score)  
 for (low, high), level in qualitative\_internal\_scale.items():  
 if low <= score <= high:  
 return level  
 except ValueError:  
 pass # Handle non-float CVSS scores if necessary  
  
 # Fallback to tool-specific mapping if CVSS not available/valid  
 sev\_lower = str(original\_severity).lower()  
 if tool\_name == "Grype":  
 if sev\_lower == "critical": return "Critical"  
 if sev\_lower == "high": return "High"  
 if sev\_lower == "medium": return "Medium"  
 if sev\_lower == "low": return "Low"  
 if sev\_lower == "negligible": return "Informational"  
 elif tool\_name == "Semgrep":  
 if sev\_lower == "error": return "High"  
 if sev\_lower == "warning": return "Medium"  
 if sev\_lower == "info": return "Informational"  
 # Add mappings for other tools...  
  
 # Default if no mapping found  
 return "Medium" # Or 'Unknown'  
  
# --- 3. Fingerprinting ---  
  
def generate\_fingerprint(finding: NormalizedFinding):  
 """Generates a stable fingerprint for deduplication."""  
 components =  
  
 # 1. Vulnerability Type (Normalized)  
 components.append(f"vuln\_id:{finding.vulnerability\_id or 'UNKNOWN'}")  
  
 # 2. Location (Normalized)  
 if finding.location.get("file\_path"):  
 components.append(f"path:{finding.location['file\_path']}")  
 # Add more stable location anchor if available (e.g., function name, AST hash)  
 # components.append(f"anchor:{finding.location.get('function\_name', '')}")  
 if finding.location.get("line"): # Use line only if more stable anchor isn't available  
 components.append(f"line:{finding.location['line']}")  
 elif finding.location.get("package\_name"): # For SCA findings  
 components.append(f"pkg\_name:{finding.location['package\_name']}")  
 components.append(f"pkg\_ver:{finding.location['package\_version']}")  
 components.append(f"pkg\_type:{finding.location['package\_type']}")  
  
  
 # 3. Code Context (Simplified: using description hash if no code snippet)  
 # In reality, hash a normalized code snippet or AST node if available  
 context\_data = finding.description or ""  
 context\_hash = hashlib.sha256(context\_data.encode()).hexdigest()[:16] # Short hash  
 components.append(f"ctx:{context\_hash}")  
  
 # Combine components into a canonical string  
 fingerprint\_string = "|".join(sorted(components))  
  
 # Hash the final string  
 return hashlib.sha256(fingerprint\_string.encode()).hexdigest()  
  
# --- 4. Deduplication ---  
  
def deduplicate\_finding(new\_finding: NormalizedFinding, existing\_fingerprints: dict):  
 """Checks if a finding is a duplicate based on fingerprint."""  
 new\_fingerprint = new\_finding.fingerprint  
 if not new\_fingerprint:  
 return # Cannot deduplicate without a fingerprint  
  
 if new\_fingerprint in existing\_fingerprints:  
 new\_finding.is\_duplicate = True  
 new\_finding.duplicate\_of\_finding\_id = existing\_fingerprints[new\_fingerprint]["finding\_id"]  
 # Update last\_seen of the original finding (logic handled by data store)  
 # Use the first\_seen timestamp from the original  
 new\_finding.first\_seen\_timestamp = existing\_fingerprints[new\_fingerprint]["first\_seen"]  
 else:  
 # This is the first time seeing this fingerprint  
 new\_finding.is\_duplicate = False  
 new\_finding.first\_seen\_timestamp = new\_finding.last\_seen\_timestamp  
 # Add to known fingerprints (logic handled by data store)  
  
 # ML-based deduplication could be applied here as a second pass  
 # if not new\_finding.is\_duplicate:  
 # potential\_duplicates = find\_semantic\_duplicates(new\_finding, all\_findings)  
 # if potential\_duplicates:  
 # # Handle potential duplicates (e.g., flag for review)  
 # pass  
  
# --- 5. Confidence Scoring ---  
  
def calculate\_confidence(finding: NormalizedFinding, historical\_data: dict):  
 """Calculates a confidence score based on tool, rule, and history."""  
 # Simplified example model  
 baseline\_confidence = {  
 "Semgrep": {"default": 0.7, "rule-high-precision": 0.9},  
 "Grype": {"default": 0.95}, # SCA findings often have higher baseline  
 "Bandit": {"default": 0.6},  
 # Add other tools  
 }  
  
 tool = finding.tool\_name  
 rule = finding.vulnerability\_id  
 base = baseline\_confidence.get(tool, {}).get(rule, baseline\_confidence.get(tool, {}).get("default", 0.5))  
  
 # Factor in historical True Positive Rate (TPR) from feedback loop  
 # historical\_data might look like: {'Semgrep': {'rule-low-precision': {'tp': 10, 'fp': 20}}}  
 stats = historical\_data.get(tool, {}).get(rule, {'tp': 1, 'fp': 0}) # Default to 100% TPR if no history  
 total = stats['tp'] + stats['fp']  
 tpr = stats['tp'] / total if total > 0 else 1.0  
  
 confidence = base \* tpr  
  
 # Adjust based on scanner confidence if provided  
 if finding.scanner\_confidence is not None:  
 confidence = (confidence + finding.scanner\_confidence) / 2 # Simple averaging  
  
 return max(0.0, min(1.0, confidence)) # Clamp between 0 and 1  
  
# --- 6. Enrichment ---  
  
def enrich\_finding(finding: NormalizedFinding):  
 """Enriches finding with external and internal context."""  
 # External TI  
 if finding.vulnerability\_id and finding.vulnerability\_id.startswith("CVE-"):  
 finding.enrichment\_data["cisa\_kev"] = finding.vulnerability\_id in CISA\_KEV\_CATALOG  
 try:  
 # In real implementation, use an HTTP client library like requests  
 # response = requests.get(EPSS\_API\_ENDPOINT.format(cve=finding.vulnerability\_id), timeout=5)  
 # epss\_data = response.json()  
 # finding.enrichment\_data["epss\_score"] = float(epss\_data["data"]["epss"])  
 # finding.enrichment\_data["epss\_percentile"] = float(epss\_data["data"]["percentile"])  
 pass # Placeholder for actual API call  
 except Exception as e:  
 print(f"Error fetching EPSS for {finding.vulnerability\_id}: {e}")  
  
 # Internal Context (simplified examples)  
 if finding.location.get("file\_path"):  
 repo\_name = finding.location["file\_path"].split('/') # Assuming path starts with repo name  
  
 # Code Owners (requires parsing CODEOWNERS file)  
 # finding.enrichment\_data["code\_owners"] = get\_code\_owners(CODEOWNERS\_PATH\_TEMPLATE.format(repo\_name=repo\_name), finding.location["file\_path"])  
  
 # Git Blame (requires running git command and parsing)  
 # blame\_info = get\_git\_blame(GIT\_REPO\_PATH\_TEMPLATE.format(repo\_name=repo\_name), finding.location["file\_path"], finding.location["line"])  
 # finding.enrichment\_data["last\_author"] = blame\_info.get("author")  
 # finding.enrichment\_data["last\_commit\_date"] = blame\_info.get("date")  
 pass # Placeholder for actual git/file operations  
  
 # Asset/CMDB Data (requires API call)  
 # asset\_name = determine\_asset\_from\_context(...) # Logic to map finding to asset  
 # if asset\_name:  
 # try:  
 # response = requests.get(CMDB\_API\_ENDPOINT.format(name=asset\_name), timeout=5)  
 # cmdb\_data = response.json()  
 # finding.enrichment\_data["asset\_criticality"] = cmdb\_data.get("criticality")  
 # finding.enrichment\_data["asset\_environment"] = cmdb\_data.get("environment")  
 # except Exception as e:  
 # print(f"Error fetching CMDB data for {asset\_name}: {e}")  
 pass # Placeholder for actual CMDB call  
  
# --- Main Processing Pipeline (Conceptual) ---  
  
def process\_scan\_report(report\_data, tool\_name, existing\_fingerprints, historical\_fp\_data):  
 """Processes a raw scan report."""  
 processed\_findings =  
 for raw\_result in report\_data: # Iterate through findings in the report  
 # 1. Normalize  
 if tool\_name == "SARIF\_Compatible\_Tool": # Assuming report\_data is parsed SARIF  
 normalized = normalize\_sarif\_finding(raw\_result, tool\_name)  
 elif tool\_name == "Grype": # Assuming report\_data is parsed Grype JSON  
 normalized = normalize\_grype\_finding(raw\_result, tool\_name)  
 # Add elif for other tools...  
 else:  
 print(f"Unsupported tool: {tool\_name}")  
 continue  
  
 # 2. Normalize Severity  
 # Extract CVSS score if available in raw\_finding  
 cvss = None # Placeholder  
 normalized.severity\_normalized = normalize\_severity(  
 normalized.severity\_original, tool\_name, cvss  
 )  
  
 # 3. Generate Fingerprint  
 normalized.fingerprint = generate\_fingerprint(normalized)  
  
 # 4. Deduplicate  
 deduplicate\_finding(normalized, existing\_fingerprints)  
  
 # 5. Calculate Confidence  
 normalized.confidence\_score = calculate\_confidence(normalized, historical\_fp\_data)  
  
 # 6. Enrich  
 enrich\_finding(normalized)  
  
 # Add to list for storage/further processing  
 processed\_findings.append(normalized)  
  
 # Persist findings to chosen data store (PostgreSQL, Elasticsearch, etc.)  
 # Update existing\_fingerprints map and historical\_fp\_data  
 #... store\_findings(processed\_findings)...  
 #... update\_fingerprint\_map(...)...  
 #... update\_historical\_data(...)...  
  
 return processed\_findings

This pseudocode illustrates the sequence of operations: normalization (schema and severity), fingerprinting, deduplication, confidence scoring, and enrichment. Each function represents a module within the framework, transforming the finding data step-by-step towards a unified, context-rich representation suitable for storage and analysis. Real-world implementation would require robust error handling, specific library choices for parsing, API interactions, and database operations.

## **4. Conclusion and Recommendations**

### **Summary of the Framework's Value**

The proliferation of security assessment tools, while beneficial for coverage, creates significant challenges in managing the resulting findings due to inconsistencies in format, severity, and location reporting. The framework detailed in this report provides a structured and comprehensive approach to address these challenges by systematically normalizing, deduplicating, enriching, and optimizing the storage of security findings.

By establishing a **common schema**, the framework creates a unified language for all findings, enabling consistent processing and analysis. **Severity normalization**, using CVSS as a baseline but allowing for contextual adjustments, permits meaningful prioritization across different tool outputs. **Resilient location mapping** and robust **fingerprinting** techniques are employed not just for display but critically for accurate **deduplication**, significantly reducing noise and redundant effort. The framework moves beyond simple detection by incorporating **confidence scoring**, refined through **feedback loops**, to assess the likelihood of a finding being a true positive, further aiding triage efficiency. Crucially, the **enrichment** component integrates external threat intelligence (CISA KEV, EPSS) and internal context (code ownership, asset criticality) to transform vulnerability management into a risk-based process. Finally, considerations for **optimized data storage and retrieval** ensure the framework can scale to handle large volumes of data effectively.

Implementing this framework allows organizations to move from a reactive, tool-siloed approach to a proactive, integrated, and risk-aware vulnerability management strategy. This leads to more efficient use of security and development resources, faster remediation of critical issues, reduced alert fatigue, and an overall stronger security posture.

### **Implementation Considerations**

Successfully implementing this framework requires careful planning and execution:

1. **Phased Approach:** Implement the framework incrementally. Start with the core components: common schema definition, ingestion adapters for key tools, severity normalization, and basic fingerprint-based deduplication. Subsequently, introduce enrichment capabilities (KEV, EPSS, CMDB), confidence scoring, the feedback loop for false positive reduction, and finally, advanced ML-based deduplication if needed.
2. **Tooling Selection:** Choose appropriate technologies.
   * *Database:* Select a database (or combination) based on primary query needs (e.g., Elasticsearch for dashboarding/search, PostgreSQL for structured reporting).39 Ensure the chosen solution can handle the expected data volume and query load.
   * *Processing Engine:* Python is a strong candidate due to its extensive libraries for data processing, ML, API interaction, and database connectivity. Consider task queues (like Celery) for asynchronous processing of enrichment or ML tasks.
   * *AST Parsing:* Select robust AST parsing libraries suitable for the primary languages in use within the organization.
3. **Schema Definition:** Invest time in defining the common schema. While the proposed schema provides a baseline, tailor it to include organization-specific fields necessary for reporting and workflow integration. Ensure it remains extensible.
4. **Normalization Logic:** Carefully define and document the severity mapping rules for each integrated tool. This requires understanding how each tool defines its severity levels.
5. **Fingerprinting Tuning:** The effectiveness of deduplication hinges on the fingerprinting strategy. Expect to tune the components included in the fingerprint based on observed deduplication accuracy and the types of code changes common in the environment.
6. **Feedback Loop Integration:** Design the user interface or API endpoints to easily capture triage feedback (True Positive, False Positive, Reason). Ensure this data is reliably processed to update confidence scores and suppression lists.
7. **Enrichment Integration:** Building reliable integrations with external APIs (EPSS, TI Feeds) and internal systems (CMDB, Git repositories, CODEOWNERS) requires effort and maintenance. Handle API rate limits and errors gracefully.
8. **Ongoing Maintenance:** The framework is not static. New tools will need adapters, normalization rules may need updates, ML models require retraining, threat feeds change, and fingerprinting logic might need refinement based on performance. Allocate resources for ongoing maintenance and improvement.

### **Future Enhancements**

Once the core framework is established, several enhancements can further increase its value:

1. **Automated Remediation Workflows:** Integrate the prioritized and enriched findings with orchestration platforms (e.g., SOAR) or ticketing systems (e.g., Jira 114) to automatically generate remediation tasks, assign them based on code ownership, and track progress.
2. **Attack Path Analysis:** Leverage graph database capabilities (if chosen or added) to model relationships between findings, assets, identities, and network paths, enabling the identification and prioritization of vulnerabilities that lie on critical attack paths.148
3. **Predictive Vulnerability Management:** Utilize historical finding data, enrichment context, and ML models to predict future high-risk areas or vulnerabilities likely to become exploited.
4. **Integration with Attack Surface Management (ASM):** Correlate findings with ASM data to better understand the external exposure associated with vulnerabilities.
5. **Advanced Visualization:** Develop dashboards that go beyond simple counts, visualizing trends, risk exposure across business units, and the effectiveness of remediation efforts over time. Graph visualizations could illustrate finding relationships and dependencies.
6. **VEX Generation:** Automatically generate VEX documents based on the analysis, enrichment, and triage status stored within the framework, facilitating communication with downstream consumers.

By implementing the core components and considering these future enhancements, organizations can build a powerful, centralized system for managing security findings, transforming raw tool output into actionable, risk-based intelligence.

#### Works cited

1. Source Code Analysis Tools | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-community/Source_Code_Analysis_Tools>
2. OWASP Developer Guide | DefectDojo, accessed May 8, 2025, <https://owasp.org/www-project-developer-guide/draft/verification/vulnerability_management/defectdojo/>
3. SAST vs. DAST - GitLab, accessed May 8, 2025, <https://about.gitlab.com/topics/devsecops/sast-vs-dast/>
4. SAST vs DAST: What they are and when to use them - CircleCI, accessed May 8, 2025, <https://circleci.com/blog/sast-vs-dast-when-to-use-them/>
5. Severity Levels for Security Issues | Atlassian, accessed May 8, 2025, <https://www.atlassian.com/trust/security/security-severity-levels>
6. Common Vulnerability Scoring System (CVSS) Value - Ivanti, accessed May 8, 2025, <https://www.ivanti.com/blog/common-vulnerability-scoring-system-cvss>
7. Denormalized vs. Normalized Data - Pure Storage Blog, accessed May 8, 2025, <https://blog.purestorage.com/purely-educational/denormalized-vs-normalized-data/>
8. How Data Deduplication Improves Investigations? - Data Ladder, accessed May 8, 2025, <https://dataladder.com/how-data-deduplication-improves-investigations/?imz_s=sbaksln69j940ble6eatuom730/>
9. About Deduplication | DefectDojo Documentation, accessed May 8, 2025, <https://docs.defectdojo.com/en/working_with_findings/finding_deduplication/about_deduplication/>
10. False Positive Elimination: Enhancing Cybersecurity Efficiency ..., accessed May 8, 2025, <https://brandefense.io/blog/drps/false-positive-elimination-enhancing-cybersecurity-efficiency/>
11. How Data Deduplication Improves Investigations?, accessed May 8, 2025, <https://dataladder.com/how-data-deduplication-improves-investigations/?imz_s=momauala65sukttjomnalonn56/>
12. What is Static Analysis Results Interchange Format (SARIF)? | Aptori ..., accessed May 8, 2025, <https://www.aptori.com/glossary/static-analysis-results-interchange-format-sarif>
13. SARIF Can't Save You Now - Boost Security, accessed May 8, 2025, <https://boostsecurity.io/blog/sarif-cant-save-you-now>
14. SARIF support for code scanning - GitHub Docs, accessed May 8, 2025, <https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning>
15. Static Analysis Results Interchange Format (SARIF) Version 2.1.0, accessed May 8, 2025, <https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html>
16. Static Analysis Results Interchange Format (SARIF) Version 2.1.0, accessed May 8, 2025, <https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html#_Toc34317648>
17. sarif-rs | SARIF CLI tools, accessed May 8, 2025, <https://psastras.github.io/sarif-rs/>
18. sarif-schema-2.1.0.json - GitHub, accessed May 8, 2025, <https://github.com/oasis-tcs/sarif-spec/blob/main/sarif-2.1/schema/sarif-schema-2.1.0.json>
19. Evaluating Macie findings with AWS Security Hub - Amazon Macie, accessed May 8, 2025, <https://docs.aws.amazon.com/macie/latest/user/securityhub-integration.html>
20. AWS Security Hub | Panther Docs, accessed May 8, 2025, <https://docs.panther.com/data-onboarding/supported-logs/aws/security-hub>
21. Optional top-level ASFF attributes - AWS Security Hub, accessed May 8, 2025, <https://docs.aws.amazon.com/securityhub/latest/userguide/asff-top-level-attributes.html>
22. Security Use Case: Vulnerability Exploitability - CycloneDX, accessed May 8, 2025, <https://cyclonedx.org/use-cases/vulnerability-exploitability/>
23. Vulnerability Exploitability eXchange (VEX) - CycloneDX, accessed May 8, 2025, <https://cyclonedx.org/capabilities/vex/>
24. CycloneDX Bill of Materials Standard | CycloneDX, accessed May 8, 2025, <https://cyclonedx.org/>
25. specification/schema/bom-1.6.schema.json at master · CycloneDX/specification - GitHub, accessed May 8, 2025, <https://github.com/CycloneDX/specification/blob/master/schema/bom-1.6.schema.json>
26. Specification [#section] - JSON Schema, accessed May 8, 2025, <https://json-schema.org/specification>
27. How CycloneDX VEX Makes Your SBOM Useful | Blog - Endor Labs, accessed May 8, 2025, <https://www.endorlabs.com/learn/how-cyclonedx-vex-makes-your-sbom-useful>
28. vex2pdf - crates.io: Rust Package Registry, accessed May 8, 2025, <https://crates.io/crates/vex2pdf>
29. vex2pdf - crates.io: Rust Package Registry, accessed May 8, 2025, <https://crates.io/crates/vex2pdf/0.4.3>
30. accessed December 31, 1969, <https://cyclonedx.org/docs/1.5/json/#components_items_vulnerabilities_items_analysis>
31. accessed December 31, 1969, <https://cyclonedx.org/docs/1.4/json/#components_items_vulnerabilities_items_analysis>
32. accessed December 31, 1969, <https://cyclonedx.org/docs/1.5/json/#components_items_vulnerabilities_items_analysis_state>
33. accessed December 31, 1969, <https://cyclonedx.org/docs/1.5/json/#components_items_vulnerabilities_items_analysis_justification>
34. accessed December 31, 1969, <https://cyclonedx.org/docs/1.5/json/#components_items_vulnerabilities_items_analysis_responses>
35. accessed December 31, 1969, <https://cyclonedx.org/docs/1.5/json/#components_items_vulnerabilities_items_analysis_detail>
36. accessed December 31, 1969, <https://cyclonedx.org/docs/1.5/json/#vulnerabilities_items_analysis>
37. accessed December 31, 1969, <https://cyclonedx.org/docs/1.5/json/>
38. PostgreSQL Indexing Best Practices Guide - MyDBOPS, accessed May 8, 2025, <https://www.mydbops.com/blog/postgresql-indexing-best-practices-guide>
39. Compare Elasticsearch vs PostgreSQL - InfluxDB, accessed May 8, 2025, <https://www.influxdata.com/comparison/elasticsearch-vs-postgres/>
40. Taking platform support for CVSS to the next level - Bugcrowd, accessed May 8, 2025, <https://www.bugcrowd.com/blog/taking-platform-support-for-cvss-to-the-next-level/>
41. What is the Common Vulnerability Scoring System (CVSS)? - Balbix, accessed May 8, 2025, <https://www.balbix.com/insights/understanding-cvss-scores/>
42. Assess severity of a vulnerability - DigiCert documentation, accessed May 8, 2025, <https://docs.digicert.com/en/software-trust-manager/threat-detection/best-practices-for-common-vulnerabilities-and-exposures/assess-the-risk-of-a-vulnerability.html>
43. Known Exploited Vulnerabilities Catalog | CISA, accessed May 8, 2025, <https://www.cisa.gov/known-exploited-vulnerabilities-catalog>
44. EPSS Scores in the GitHub Advisory Database - GitHub Changelog, accessed May 8, 2025, <https://github.blog/changelog/2024-10-10-epss-scores-in-the-github-advisory-database/>
45. CVSS v3.1 Specification Document - FIRST.org, accessed May 8, 2025, <https://www.first.org/cvss/v3.1/specification-document>
46. accessed December 31, 1969, <https://blog.sonarsource.com/accurate-issue-tracking-for-static-analysis/>
47. accessed December 31, 1969, <https://dl.acm.org/doi/10.1145/3377811.3382371>
48. accessed December 31, 1969, <https://dl.acm.org/doi/pdf/10.1145/3337970.3337971>
49. accessed December 31, 1969, <https://www.researchgate.net/publication/351648095_A_Robust_Approach_for_Tracking_Static_Analysis_Warnings_Across_Software_Versions>
50. accessed December 31, 1969, <https://www.researchgate.net/publication/320293850_AST-Based_Technique_for_Tracking_Static_Analysis_Alerts_Through_Code_Transformations>
51. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/8632589>
52. Vulnerability Detection in Source Code via Subgraph Isomorphism Analysis, accessed May 8, 2025, <https://wpcdn.web.wsu.edu/wp-vcea/uploads/sites/3267/2024/05/Vulnerability-Detection-in-Source-Code-via-Subgraph-Isomorphism-Analysis.pdf>
53. General | Developer Tools | Abstract Syntax Tree - Codecademy, accessed May 8, 2025, <https://www.codecademy.com/resources/docs/general/developer-tools/abstract-syntax-tree>
54. (PDF) Generalized Vulnerability Extrapolation using Abstract Syntax Trees - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/233997025_Generalized_Vulnerability_Extrapolation_using_Abstract_Syntax_Trees>
55. Generalized Vulnerability Extrapolation using Abstract Syntax Trees - MLSEC, accessed May 8, 2025, <https://mlsec.tu-berlin.de/docs/2012-acsac.pdf>
56. accessed December 31, 1969, <https://engineering.grammarly.com/blog/managing-code-locations-at-scale-with-asts-and-other-techniques/>
57. Fingerprint (computing) - Wikipedia, accessed May 8, 2025, <https://en.wikipedia.org/wiki/Fingerprint_(computing)>
58. VulMatch: Binary-Level Vulnerability Detection Through Signature - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/389844102_VulMatch_Binary-Level_Vulnerability_Detection_Through_Signature>
59. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities, accessed May 8, 2025, <https://www.researchgate.net/publication/365230735_TRACER_Signature-based_Static_Analysis_for_Detecting_Recurring_Vulnerabilities>
60. Confidence Scoring in Threat Intelligence | Cyware, accessed May 8, 2025, <https://www.cyware.com/resources/security-guides/cyber-threat-intelligence/what-is-confidence-scoring-in-threat-intelligence>
61. What is the vulnerability confidence rating and why is it important ..., accessed May 8, 2025, <https://www.acunetix.com/support/docs/faqs/what-is-vulnerability-confidence-rating/>
62. Source Code Analysis Tools | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-community/Source_Code_Analysis_Tools#Assessing_Confidence>
63. accessed December 31, 1969, <https://www.researchgate.net/publication/344061289_A_Methodology_for_Evaluating_and_Comparing_Static_Application_Security_Testing_SAST_Tools_Effectiveness_in_Detecting_Vulnerabilities_and_Reporting_False_PositivesNegatives>
64. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/8418609>
65. accessed December 31, 1969, <https://www.researchgate.net/publication/334890090_Confidence_Levels_in_Static_Analysis_Tools_A_Case_Study_with_FindBugs_and_SpotBugs>
66. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9191161>
67. Evaluating SAST Tools | Praetorian, accessed May 8, 2025, <https://www.praetorian.com/blog/evaluating-sast-tools/>
68. SAST vs. DAST: Understanding the Difference - Legit Security, accessed May 8, 2025, <https://www.legitsecurity.com/blog/sast-vs-dast>
69. accessed December 31, 1969, <https://www.opswat.com/blog/false-positives-what-they-are-and-how-to-minimize-them>
70. accessed December 31, 1969, <https://www.heavy.ai/blog/best-practices-for-reducing-false-positives-in-your-security-environment>
71. accessed December 31, 1969, <https://www.recordedfuture.com/blog/closing-the-loop-using-feedback-to-improve-security-alerting>
72. accessed December 31, 1969, <https://www.nist.gov/itl/ssd/software-quality-group/national-vulnerability-database-nvd/nvd-program-transition-information#FP%20Mitigation>
73. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9404098>
74. accessed December 31, 1969, <https://www.researchgate.net/publication/330400930_Reducing_False_Positives_in_Static_Analysis_through_Developer_Feedback_and_Data_Mining>
75. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/8935301>
76. Digital Fingerprinting in Cybersecurity: OS, Nmap, & More, accessed May 8, 2025, <https://www.bitsight.com/learn/digital-fingerprinting>
77. Fingerprinting Vulnerabilities - Veracode, accessed May 8, 2025, <https://www.veracode.com/security/fingerprinting-vulnerabilities>
78. What is Fingerprinting in Cybersecurity? - Recorded Future, accessed May 8, 2025, <https://www.recordedfuture.com/threat-intelligence-101/vulnerability-management-threat-hunting/fingerprinting-in-cybersecurity>
79. Hash Value: The Fingerprint of Data: Using Hash Values to Verify Digital Evidence, accessed May 8, 2025, <https://fastercapital.com/content/Hash-Value--The-Fingerprint-of-Data--Using-Hash-Values-to-Verify-Digital-Evidence.html>
80. Fingerprint-Based Data Deduplication Using a Mathematical Bounded Linear Hash Function, accessed May 8, 2025, <https://www.mdpi.com/2073-8994/13/11/1978>
81. CWE-328: Use of Weak Hash (4.17), accessed May 8, 2025, <https://cwe.mitre.org/data/definitions/328.html>
82. accessed December 31, 1969, <https://support.sonatype.com/hc/en-us/articles/360051364554-Understanding-how-Sonatype-Lifecycle-handles-duplicate-vulnerabilities>
83. accessed December 31, 1969, <https://resources.contrastsecurity.com/hubfs/Understanding_Vulnerability_Fingerprinting_A_Developers_Guide_to_Deduplication_and_Tracking.pdf>
84. accessed December 31, 1969, <https://www.researchgate.net/publication/332322491_Fingerprinting_Software_Vulnerabilities_A_Systematic_Mapping_Study>
85. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9327640>
86. accessed December 31, 1969, <https://www.researchgate.net/publication/344560089_VulDedupe_A_Vulnerability_Deduplication_System_for_Open-Source_Software_Supply_Chains>
87. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9463095>
88. Include fingerprint in SARIF output · Issue #1058 - GitHub, accessed May 8, 2025, <https://github.com/hadolint/hadolint/issues/1058>
89. accessed December 31, 1969, <https://github.com/microsoft/sarif-sdk/issues/1875>
90. accessed December 31, 1969, <https://sarifweb.azurewebsites.net/sarif-wiki/Fingerprinting.html>
91. SARIF support for code scanning - GitHub Docs, accessed May 8, 2025, <https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning#providing-data-to-track-code-scanning-alerts-across-runs>
92. accessed December 31, 1969, <https://www.researchgate.net/publication/343200013_SARIF_Static_Analysis_Results_Interchange_Format_A_Standard_for_Static_Analysis_Tool_Output>
93. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9607771>
94. FED: Fast and Efficient Dataset Deduplication Framework with GPU Acceleration - arXiv, accessed May 8, 2025, <https://arxiv.org/html/2501.01046v1>
95. accessed December 31, 1969, <https://www.securitybridge.com/news/the-art-of-deduplication-how-to-effectively-manage-security-vulnerabilities.html>
96. accessed December 31, 1969, <https://www.researchgate.net/publication/369638950_A_Systematic_Literature_Review_on_Vulnerability_Deduplication_Techniques>
97. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9662400>
98. accessed December 31, 1969, <https://www.researchgate.net/publication/359099360_Vulnerability_Prioritization_A_Systematic_Mapping_Study_and_Future_Research_Agenda>
99. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9796575>
100. Safe C++, accessed May 8, 2025, <https://safecpp.org/>
101. Privacy-preserving deduplication to enhance federated learning, accessed May 8, 2025, <https://from.ncl.ac.uk/privacy-preserving-deduplication-to-enhance-federated-learning>
102. fdtn-ai/Foundation-Sec-8B - Hugging Face, accessed May 8, 2025, <https://huggingface.co/fdtn-ai/Foundation-Sec-8B>
103. Reduce Noise With Alert Deduplication - ilert, accessed May 8, 2025, <https://www.ilert.com/ai-incident-management-guide/reduce-noise-with-alert-deduplication>
104. (PDF) AI-Driven Categorization and Deduplication - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/389210560_AI-Driven_Categorization_and_Deduplication>
105. (PDF) The Role of Natural Language Processing in Cybersecurity - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/387252722_The_Role_of_Natural_Language_Processing_in_Cybersecurity>
106. accessed December 31, 1969, <https://ieeexplore.ieee.org/document/9796575>
107. accessed December 31, 1969, <https://www.elastic.co/blog/machine-learning-categorization-nlp-elasticsearch>
108. www.aclweb.org, accessed May 8, 2025, <https://www.aclweb.org/anthology/2020.lrec-1.308.pdf>
109. accessed December 31, 1969, <https://www.researchgate.net/publication/348500944_An_Empirical_Study_on_the_Use_of_Machine_Learning_for_Vulnerability_Report_Deduplication>
110. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9191166>
111. accessed December 31, 1969, <https://www.researchgate.net/publication/335724080_DeepVD_A_Deep_Learning_Based_Approach_for_Vulnerability_Description_Classification_and_Similarity_Analysis>
112. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9040590>
113. A SURVEY ON THE USE OF DATA CLUSTERING FOR INTRUSION DETECTION SYSTEM IN CYBERSECURITY - PMC, accessed May 8, 2025, <https://pmc.ncbi.nlm.nih.gov/articles/PMC8289996/>
114. OWASP Defectdojo – OWASP Nest, accessed May 8, 2025, <https://nest.owasp.org/projects/defectdojo>
115. Manage Duplicate Findings - DefectDojo Documentation, accessed May 8, 2025, <https://docs.defectdojo.com/en/working_with_findings/findings_workflows/manage_duplicate_findings/>
116. Using Reachability & Exploitability to Reduce CVE False Positives - OpsMx, accessed May 8, 2025, <https://www.opsmx.com/blog/reducing-cve-false-positives-using-reachability-and-exploitability/>
117. Reduce False Positives in AML: Best Practices and Examples - FOCAL, accessed May 8, 2025, <https://www.getfocal.ai/blog/reduce-false-positives-in-aml>
118. Review | AquilaX Docs, accessed May 8, 2025, <https://docs.aquilax.ai/ai-models/review>
119. What Is Alert Triage in Cybersecurity? Steps, Challenges, and Tips | Corelight, accessed May 8, 2025, <https://corelight.com/resources/glossary/alert-triage>
120. Charlotte AI: Generative AI for Cybersecurity | CrowdStrike, accessed May 8, 2025, <https://www.crowdstrike.com/en-us/platform/charlotte-ai/>
121. Contextual Risk Estimation for Effective Prioritization | Microsoft ..., accessed May 8, 2025, <https://techcommunity.microsoft.com/t5/microsoft-defender-for-cloud/contextual-risk-estimation-for-effective-prioritization/ba-p/4035425>
122. Leveraging Data Enrichment in Cyber Threat Intelligence (CTI ..., accessed May 8, 2025, <https://www.anomali.com/blog/leveraging-data-enrichment-in-threat-intelligence>
123. Reducing the Significant Risk of Known Exploited Vulnerabilities - CISA, accessed May 8, 2025, <https://www.cisa.gov/known-exploited-vulnerabilities>
124. Prioritizing Cybersecurity with the EPSS Lookup Tool: A Deep Dive into Version 1.5, accessed May 8, 2025, <https://www.youtube.com/watch?v=pHr1muJfI4Q>
125. Exploit Prediction Scoring System (EPSS), accessed May 8, 2025, <https://www.first.org/epss/api>
126. accessed December 31, 1969, <https://www.paloaltonetworks.com/blog/prisma-cloud/contextual-security-prisma-cloud/>
127. accessed December 31, 1969, <https://www.rezonate.io/blog/unlocking-the-power-of-context-in-cloud-security-a-primer-on-contextualization/>
128. accessed December 31, 1969, <https://docs.gitlab.com/ee/user/project/code_owners/>
129. The EPSS Model, accessed May 8, 2025, <https://www.first.org/epss/model>
130. accessed December 31, 1969, <https://www.researchgate.net/publication/360360078_Prioritizing_Software_Vulnerabilities_Using_Machine_Learning_and_Expert_Knowledge_A_Case_Study_of_EPSS>
131. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9714431>
132. git-blame Documentation - Git, accessed May 8, 2025, <https://git-scm.com/docs/git-blame>
133. accessed December 31, 1969, <https://www.servicenow.com/products/vulnerability-response/how-it-works.html>
134. CyberSecurity Asset Management 3.0 (CSAM) | Qualys, accessed May 8, 2025, <https://www.qualys.com/apps/cybersecurity-asset-management/>
135. Implementing: Storing and Securing Your Data - Research Data ..., accessed May 8, 2025, <https://guides.lib.uw.edu/research/dmg/storage-security>
136. Secure Data Storage: The 6 Best Practices | Object First, accessed May 8, 2025, <https://objectfirst.com/guides/data-storage/secure-data-storage/>
137. Can Neo4j Replace PostgreSQL in Healthcare? - PMC, accessed May 8, 2025, <https://pmc.ncbi.nlm.nih.gov/articles/PMC7233060/>
138. Optimal Indexing Strategies for PostgreSQL: Boosting Performance and Scalability, accessed May 8, 2025, <https://speakerdeck.com/shiviyer007/optimal-indexing-strategies-for-postgresql-boosting-performance-and-scalability>
139. Documentation: 17: CREATE INDEX - PostgreSQL, accessed May 8, 2025, <https://www.postgresql.org/docs/current/sql-createindex.html>
140. accessed December 31, 1969, <https://www.enterprisedb.com/blog/postgresql-performance-tuning-designing-database-schemas-maximum-efficiency>
141. accessed December 31, 1969, <https://www.cybertec-postgresql.com/en/indexing-strategies-for-optimal-postgresql-query-performance/>
142. Elasticsearch vs Graph Database | e-Zest, accessed May 8, 2025, <https://blog.e-zest.com/elasticsearch-vs-graph-database/>
143. modelcontextprotocol/servers: Model Context Protocol Servers - GitHub, accessed May 8, 2025, <https://github.com/modelcontextprotocol/servers>
144. Elasticsearch vs Neo4j on Vector Search Capabilities - Zilliz blog, accessed May 8, 2025, <https://zilliz.com/blog/elasticsearch-vs-neo4j-a-comprehensive-vector-database-comparison>
145. Tune for search speed | Elastic Docs, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-search-speed.html>
146. accessed December 31, 1969, <https://www.elastic.co/blog/structuring-data-elasticsearch-object-nested-parent-child>
147. Vector database vs. graph database: Understanding the differences | Elastic Blog, accessed May 8, 2025, <https://www.elastic.co/blog/vector-database-vs-graph-database>
148. accessed December 31, 1969, <https://neo4j.com/blog/knowledge-graphs-cybersecurity-graphcipher/>
149. accessed December 31, 1969, <https://medium.com/neo4j/graph-databases-for-cybersecurity-enhancing-threat-detection-and-response-9a727c73cf9c>
150. Mapping | Elastic Docs, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html>
151. Get Security index settings | Elasticsearch Guide [8.18] | Elastic, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/8.18/security-api-get-settings.html>
152. Update Security index settings | Elasticsearch Guide [8.18] | Elastic, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/8.18/security-api-update-settings.html>